Parlog as a Systems Programming Language

Ian Foster

Imperial College
of Science & Technology

Department of
Computing

Parlog as a Systems Programming Language

Ian Foster

Research Report PAR 88/5

March 1988

Department of Computing
Imperial College of Science and Technology
University of London
180 Queens Gate
London SW7 2BZ

Telephone: 01-589 5111

Parlog as a Systems Programming Language

Ian Foster

A thesis submitted to the University of London
for the degree of Doctor of Philosophy.

Department of Computing
Imperial College of Science and Technology

March 1988

Abstract

High-level languages have proved to be an excellent tool for dealing with complexity in operating system design. This work explores the use of *declarative* languages for systems programming. Advantages claimed for these languages include their very high-level nature and their suitability for implementation on parallel architectures. However, many declarative languages are too abstract to permit effective control of a machine. Also, it is not clear how declarative language-based operating systems are to be constructed.

The main objective of the research reported herein is the development of a methodology for the design and implementation of declarative language-based operating systems. For concreteness, this is based on a particular language: the parallel logic programming language Parlog. The methodology views a Parlog operating system as a layered structure. A machine language or hardware kernel supports the Parlog language. Operating system services are implemented in Parlog. User-level programs such as programming environments are constructed using these services. The methodology provides simple, coherent treatments of important issues in the design of each layer in this structure.

A Parlog operating system kernel is defined in terms of a simple 'kernel language', a subset of Parlog. An implementation scheme for this kernel language is described and its efficiency is confirmed by experimental studies. It is shown that important operating system components can be implemented in the kernel language.

A comprehensive set of techniques for the programming of Parlog operating systems, on uniprocessors and multiprocessors, is presented. New Parlog language features required for systems programming are introduced.

Operating system support for programming environment implementation and application programming is defined in terms of an extended Parlog language named Parlog+. Parlog+ extends Parlog with metaprogramming facilities that permit programs to access and specify changes to program files. The application of Parlog+ and its implementation in Parlog are described.

Acknowledgements

First of all, I would like to thank my advisor, Keith Clark, for his help, encouragement and support throughout the course of this research.

I would like to thank the people who aided in the development of the Parlog implementation that made the experimental component of this research possible: Alastair Burt, Martyn Cutcher, David Gilbert, Tony Kusalik, Graem Ringwood and Ken Satoh. More generally, I should thank all of my colleagues at Imperial College, especially Jim Crammond, Andrew Davison, Chris Hogger, Melissa Lam and Frank McCabe. The administrative support provided by Cheryl Anderson, Caroline Wraige and the logic programming group administrator, Pat Easton-Orr, is gratefully acknowledged.

I am grateful to Yonit Keston, Jeff Magee and John Darlington for reading drafts of this thesis and providing useful comments. Steve Gregory's particularly rigorous perusal and detailed comments were especially valuable.

Visits to other institutions have proved very beneficial to this research. I would like to thank Ehud Shapiro, of the Weizmann Institute, and Kazuhiro Fuchi and Koichi Furukawa, of the ICOT research centre, for making these visits possible. I would also like to thank the many people who made these visits fruitful and enjoyable; in particular, Steve Taylor and Jiro Tanaka.

My research has been funded by the Science and Engineering Research Council.

Contents

14

List of Figures

List of Tables

List of Programs

CHAPTER 1.

Introduction

1.1 Background

Systems programming — the writing of programs to control computer systems — is difficult. The systems programmer must deal with issues such as asynchrony, non-determinism and concurrency. Difficulties are compounded by the complexity of systems and the frequent requirement for extreme reliability. The operating systems that control large mainframe computers are among the most complex artifacts ever constructed by man.

Many methodologies have been proposed to ease the tasks of operating system design and implementation. High-level programming languages have proved to be among the most useful.

1.1.1 Language-Based Operating Systems

Dijkstra was one of the first to advocate a systematic solution to systems programming problems. He proposed a layered approach to operating system design. Problematic aspects of computer systems are encapsulated inside simple abstractions supported by the lower levels of an operating system. For example, non-determinism due to interrupts may be dealt with by introducing multiple sequential threads of control or **processes**. Higher levels can then ignore the existence of interrupts.

Dijkstra's classic THE operating system [1968a] showed that a structured approach to operating system design can significantly reduce system complexity. However, THE was implemented in machine language. Its structure was an artificially imposed discipline: there was no guarantee that abstractions would be used correctly or at all. A promising solution to this problem is to incorporate abstractions such as processes in a high-level language. This permits compile-time checking of program correctness and moves the burden of implementing abstractions from the systems programmer to the compiler writer. Operating systems implemented in a high-level language may be termed **language-based operating systems**, as their structure is derived from (and constrained by) the language in which they are written. Language-based operating systems generally provide a small machine-coded or hardware **kernel** which supports

programming in the high-level language. The rest of the operating system is implemented entirely in the high level language.

A number of implementation projects have demonstrated the advantages of a language-based approach to operating system design [Brinch Hansen, 1976; Joseph *et al.*, 1984; Swinehart *et al.*, 1986].

1.1.2 Symbolic Processing and Parallel Computers

The high-level languages used in the operating system implementation projects mentioned above provide the programmer with abstractions such as recursion and processes that are not found in machine languages. However, they remain *machine-oriented* or *imperative* languages. That is, they derive their structure from the (von Neumann) computer they are intended to execute on. Programs in these languages specify changes to the state of a machine. These are translated more or less directly into sequences of instructions to be executed by a computer.

Imperative languages can be implemented efficiently on von Neumann computers. The numerical and data processing applications that these computers support are generally implemented in the same sort of language. A reliance on imperative languages as systems programming languages is hence understandable. However, developments in computer applications and architectures are beginning to challenge the dominant position that imperative languages and the von Neumann computer occupy in computing.

As computers become cheaper and understanding of how to use them improves, there is an increasing tendency to apply them in more abstract and 'human-oriented' fields (for example, see [Kawanobe, 1984]). Applications in these fields — such as knowledge bases, computer aided design and natural language understanding — are generally concerned with manipulating symbols rather than numbers. Such *symbolic processing* applications can be expected to become increasingly important in the future.

The development of *parallel computers* is spurred by a need for greater performance at a reasonable price. Computer engineering is approaching the limits of the performance that can be achieved using a single von Neumann processor. A consequence of this is the intense research (and recently, commercial exploitation) of alternative architectures based on tens, hundreds or thousands of processors (for example, see [Vegdahl, 1984]).

Both symbolic computing and parallel computers present problems for the systems programmer. Machine-oriented languages such as those used to implement the language-based operating systems referenced above can manipulate numbers and simple data structures such as strings and arrays. However, they are in general ill-suited to manipulating symbols.

Furthermore, these languages, whilst good at implementing uniprocessor operating systems, are less well-suited to parallel machines. Programs in these languages specify sequences of changes to the state of a machine. This reliance on *side-effects* means that these languages are inherently sequential. Parallelism *can* be introduced, as in languages such as Concurrent Pascal and CSP, but only in a manner orthogonal to the basic sequential model of computation. The resulting distinction between sequential and parallel execution places a heavy intellectual burden on the programmer, who is in effect forced to think in two dimensions. This can become very difficult on a multiprocessor, as the number of parallel processes tends to grow at least proportionally to the number of processors.

1.1.3 Declarative Languages

Not all computer languages share the difficulties inherent in the machine-oriented or imperative languages. Another class of languages, the declarative languages, has quite different characteristics. These languages are based on abstract, generally symbolic formalisms rather than a computer architecture. Ideally, they permit the programmer to describe the problem to be solved, rather than what the machine must do to solve it. They do not rely on side-effects to perform computation.

Declarative approaches to programming include functional programming, based on a mathematical theory of functions, and logic programming, based on a restricted form of first order logic. Programs in declarative formalisms can be read declaratively as statements about a problem domain. They can also be executed, using an appropriate evaluation strategy, to perform computation. This is the basis for their use as programming languages.

A declarative programming language consists of a declarative formalism plus an evaluation strategy. For example, a logic program consists of logic clauses. These can be read declaratively as definitions of relationships between entities. They can also, as Kowalski [1974] showed, be read as procedures for computing instances of the relations defined by the program. Prolog [Roussel, 1975], the best known logic

programming language, uses a sequential evaluation strategy based on resolution and depth-first, backtracking search to execute logic programs and hence perform this computation. A Prolog program that defines the 'sort' relation reads declaratively as the definition of a relation between two lists, one of which is a sorted version of the other. This program can also be executed to sort lists.

The declarative content of declarative language programs is claimed to have significant advantages from a software engineering point of view [Backus, 1978; Henderson, 1980; Kowalski, 1983]. The correspondence of the solutions computed by a language and those implied by a program's declarative reading permits programs to be read as specifications. Systematic synthesis and transformation techniques can be applied to programs. Declarative content also facilitates program development, modification and reuse.

Two other characteristics of these languages are particularly relevant to the systems programmer. First, they support symbolic manipulation. Logic programming languages, for example, support fully recursively defined data structures and provide the powerful bidirectional matching operation, unification.

Second, they are well-suited to parallel evaluation. Parallelism is not orthogonal to these languages but implicit in their computational models. Their lack of side-effects means that the declarative language programmer is obliged neither to needlessly sequence operations nor to specify when they may be executed in parallel. Instead, he provides an abstract specification of a problem. Implicit parallelism in this specification can be detected and exploited by a language's evaluation strategy. In some declarative languages, a separate control language permits the programmer to control how parallelism is exploited. This permits the specification of parallel algorithms.

1.1.4 Declarative Language-Based Operating Systems

The declarative reading, high-level features and implicit parallelism of declarative formalisms appear to offer solutions to problems posed to the systems programmer by a new generation of applications and architectures. Implicit parallelism eases the burden on the systems programmer, who is no longer required to think in 'two dimensions'. Declarative programming style and high-level features facilitate the construction of systems to support symbolic processing. This suggests that there may be advantages in using declarative languages for systems programming. This can lead to the development of what may be termed *declarative* language-based systems, which derive their structure from a declarative formalism.

The development of declarative language-based systems is not necessarily straightforward. Most declarative languages are *too abstract* or *insufficiently expressive* for use as systems programming languages. A systems programmer requires precise control over a machine and must be able to express a wide range of algorithms. Abstractions that hide machine features can be obstacles rather than aids to programming. For example, if parallelism is completely implicit in a language, and hence removed from programmer control, the systems programmer is unable to implement processor scheduling algorithms.

Nevertheless, several researchers have shown that particular declarative languages can be used for simple systems programming tasks. Henderson [1982] described an approach to the implementation of 'purely functional operating systems'. He showed that a pure functional language with a concurrent evaluation strategy can be used to implement simple operating system structures. This work exploited an evaluation strategy for functional programs incorporating **stream parallelism** — concurrent evaluation of a function and its arguments — to implement operating systems as networks of nested functions which map inputs to outputs.

Shapiro [1984a], Clark and Gregory [1984] and Kusalik [1986] apply similar techniques to program simple operating system components such as monitors and command interpreters in parallel logic languages. Parallel logic languages [Takeuchi and Furukawa, 1986] are a family of logic programming languages with an evaluation strategy that appears well-suited to systems programming. In these languages, conjunctions of goals are evaluated concurrently. Dataflow constraints ensure that certain goals incrementally *produce* and other goals incrementally *consume* bindings for shared variables. This provides what is known as **stream and-parallelism**. This permits logic programs to be interpreted as defining systems of concurrent, communicating processes. Other important features of parallel logic languages are their guarded-command non-determinacy and unification, which permits the specification of dynamic communication structures.

Existing implementations of functional and parallel logic languages do not provide the performance required to control computer systems. However, it should be noted that:

- Conventional processors are not designed to support declarative languages. Implementations of declarative languages on these machines thus frequently *emulate* a more appropriate architecture. New processor designs that implement

these architectures directly can execute declarative languages more efficiently [Tick and Warren, 1984; Uchida, 1987].

As declarative languages are further removed from conventional architectures than imperative languages, compilation of these languages is more difficult. Yet compilation techniques for declarative languages have received less attention to date. Improved compilation techniques can be expected to improve performance.

The difficulties inherent in programming massively parallel machines favour an abstract formalism that relieves the programmer of some of the burden of specifying distributed computations. Somewhat less efficient execution may be acceptable if this permits these machines to be utilized more effectively,

1.2 The Research

1.2.1 Objectives of the Research

The basic objective of the research reported herein is a methodology for the design and implementation of declarative language-based operating systems.

The form of the investigation has been determined by three basic decisions:

- The choice of implementation language.
- The choice of target architecture.
- The choice of operating system design problems the methodology is to address.

The choice of language. The form of a language-based operating system is to a large extent determined by the language in which it is implemented. An investigation into language-based systems must necessarily be based on a particular language.

A decision was made to base this research on an existing, proven declarative language, with the option of extending or modifying this language if required. The main advantages of choosing an existing language, rather than attempting to design a new language, are: (a) it permits attention to be concentrated on systems programming rather than language design problems; and (b) since implemenations exist, it enables experimentation with methodologies.

The language selected is the parallel logic programming language Parlog [Clark and Gregory, 1986]. Parlog has the following points in its favour as a language for

programming declarative language-based systems: (a) As noted above, parallel logic languages are derived from a declarative formalism, logic programming, and are capable of specifying concurrency, non-determinism, etc. (b) A uniprocessor implementation of Parlog was available for experimental studies [Foster *et al.*, 1986]. (c) Though parallel implementations of Parlog did not exist at the time this research was started, the language is designed for implementation on parallel machines [Gregory, 1987]. (d) A range of applications indicate the utility of parallel logic languages as application programming languages (see, for example, references in [Gregory, 1987; Shapiro, 1986; Ringwood, 1988]).

Other potential 'declarative systems programming languages' are reviewed in the following chapters. Many of the design and implementation techniques developed in the course of this research can also be applied to these languages.

The choice of architecture. The design of an operating system is closely linked to the architecture of the machine it is to control. To permit concrete discussion of operating system design issues, a particular class of architecture must be considered. Since a major motivation for declarative language-based systems is the problem of programming parallel computers, this should be a parallel architecture.

Many different parallel architectures have been proposed, some of which differ quite radically from conventional machines [Vegdahl, 1984; Treleaven *et al.*, 1982]. However, most of the present and next generation of parallel computers can be expected to link together a number of more or less conventional processors using a shared memory or a message passing network. The class of machine considered here may be characterized as follows:

- a finite number of nodes, connected by a reliable communication network;
- no global storage; instead, each node has local storage;
- nodes may communicate by message passing;
- each node can execute recursive programs.

In subsequent discussion, the term *multiprocessor* will be used to refer to this class of parallel architecture.

For the purposes of this research, this class of architecture represents a compromise between generality and simplicity. It represents a quite general paradigm for parallel processing. In contrast to shared-memory machines, it is scaleable. Also, implementation techniques based on message passing can easily be adapted for shared-memory execution; the reverse is not generally true. At the same time, the assumption

of uniformity and reliable communications keep problems relatively simple. Parlog may well be a suitable systems programming language for the more general class of distributed computer systems [Sloman and Kramer, 1986]. However, distributed systems raise problems such as network reliability that are beyond the scope of the present work.

The choice of design problems. This is determined to a large extent by computer architecture. The hardware architecture assumed here implies the following software architecture: a *kernel* provides run-time support for an *operating system* (written in the high-level systems programming language), which in turn supports *programming environments* and other *applications*. The design of a Parlog operating system kernel that is simple, flexible and efficient is an important component of the research. At the operating system level, it is assumed that well-known operating system design problems such as provision of services, naming, protection and resource management must be addressed [Peterson and Silberschatz, 1983; Tanenbaum and van Renesse, 1985]. The development of coherent treatments of these problems in Parlog is a second major component of the research.

It is assumed that a Parlog operating system is intended to support Parlog programming. Operating system support for application programming in declarative languages is a complex problem that is not addressed in its entirety here. Instead, one particularly important aspect, namely program update, is considered in detail. Much of a programmer's time is occupied with tasks that involve modifying the representations of programs located in a *file system*. A declarative treatment of program file update hence appears desirable, so that tools developed to perform these tasks can benefit from a declarative programming style. Ideally, it should be possible to read programs that modify program files as specifications of relations over file system states.

Given this choice of language, architecture and design problems, the criteria to be satisfied by the operating system design methodology can be summarized as follows:

1. It should provide a comprehensive, coherent treatment of important issues in operating system design.
2. It should deal with the particular problems of multiprocessors.
3. It should not require major modifications to the implementation language, Parlog.
4. It should provide a kernel that is simple, efficient and flexible and at the same time supports all functions required by an operating system.
5. It should support a declarative treatment of program file update.

6. Its techniques should be applicable to other, similar languages.
7. It should be specified in sufficient detail to permit implementation and, where appropriate, evaluation of important components.

1.2.2 Contributions

The result of the research is a methodology for the design and implementation of language-based operating systems in the parallel logic programming language Parlog. The methodology deals with three levels of operating system design: the kernel which supports the operating system, the operating system itself, and programming environments intended to execute on the operating system.

The main original contributions are:

1. To demonstrate that it is feasible to use a declarative language, Parlog, as a language for implementing language-based operating systems.
2. To extend Parlog with features required for operating system implementation and to show how important issues in operating system design can be treated in Parlog.
3. To design a Parlog operating system kernel based on a *kernel language* (a subset of Parlog, augmented with simple primitives) and to describe how Parlog language features required for systems programming can be implemented in this kernel language.
4. To provide an abstract machine implementation of the kernel language. This is presented in the form of extensions to an existing abstract machine.
5. To present distributed unification algorithms that permit multiprocessor execution of Parlog and that support distributed termination and deadlock detection.
6. To develop a new parallel logic programming language, Parlog+, that provides a declarative treatment of program file update, and to show how this language can be used to implement programming environments in a Parlog operating system.
7. To develop techniques that permit the implementation of Parlog+ in Parlog.

Of the eight chapters in this monograph, four (Chapters 4 to 7) constitute an account of original research; relevant related work is noted in the text.

1.3 Overview of Contents

The rest of this monograph consists of seven chapters. Chapters 2 and 3 are introductory in nature, Chapters 4 – 7 present the results of the research and Chapter 8 concludes.

Chapter 2 establishes requirements for languages for programming language-based operating systems. Four classes of language — imperative, actor, functional and logic — are reviewed and evaluated.

Chapter 3 introduces the parallel logic language Parlog and makes a preliminary appraisal of its suitability as a systems programming language.

Chapter 4 identifies important issues in operating system design. Minor extensions to Parlog are proposed and treatments of operating system design issues in the extended language are described. Issues covered include the provision of services, communication with the kernel, protection and naming. A framework for a simple Parlog operating system is presented.

Chapter 5 deals with the design and implementation of a Parlog operating system kernel. A simple kernel language to be supported by this kernel is defined. The implementation of this kernel language on uniprocessors and multiprocessors is described.

Chapter 6 deals with operating system design issues that are particular to multiprocessors. The implementation of distributed computation control and processor scheduling functions is described. A framework for a simple multiprocessor Parlog operating system is presented.

Chapter 7 describes a new parallel logic programming language named Parlog+. The language is motivated and described. Simple examples illustrate its application. The principles of its implementation in Parlog are described.

Chapter 8 concludes, surveys some related research and notes areas for further research

Two appendices present a specification for a Parlog language feature, its control metacall, and experimental results that permit comparison of two approaches to kernel design. These are referred to in the text..

CHAPTER 2

Languages for Systems Programming

This chapter surveys proposed high-level systems programming formalisms and languages. It is not intended to constitute an introduction to systems programming: the interested reader is referred to [Peterson and Silberschatz, 1983]. Nor is the survey intended to be comprehensive. It aims to present important concepts and to illustrate their application.

To motivate this survey, the first part of the chapter proposes requirements for a high-level systems programming language. The particular requirements of multiprocessors are emphasized. Four classes of languages are then considered. The first, **sequential processes**, is an elaboration of the usual imperative model of computation. Computation in these languages involves concurrently executing, sequential processes which communicate by shared variables or message passing. Languages in this class include Concurrent PASCAL, Occam and Ada. Computation in the **actors** formalism involves dynamic networks of communicating agents. The more abstract **functional** languages represent computation as the evaluation of mathematical functions. Functions are applied to input values and compute output values. Languages in this class include pure LISP, HOPE and SASL. Finally, **logic programming** languages model computation as controlled deduction. Languages in this class include Prolog and Parlog. Logic programming and Prolog are discussed in this chapter. Parlog is introduced in Chapter 3.

2.1 Language Requirements

Requirements for a high-level, multiprocessor systems programming language can be divided into three groups: expressiveness, efficiency and utility.

These requirements are not complete; for example, issues related to reliability are not considered. However, they cover most important issues in systems programming language design.

2.1.1 Expressiveness

Concurrency

The concurrent process is an important abstraction in systems programming, where it represents a logically distinct thread of control [Dijkstra, 1968a]. If programs are to execute faster on multiprocessors, it must also be possible to divide them into components that execute on different nodes. A high-level systems programming language should permit the specification of logically and/or physically distinct processes.

Communication and Synchronization

In order to cooperate, concurrent processes must be able to exchange data: that is, to communicate. They must also be able to synchronize their execution so as to order their activities [Andrews and Schneider, 1983]. For example, to ensure that a buffer is not read before it is written.

Non-determinism

The behaviour of systems programs must often depend on the timing and ordering of events. For example, interrupts or communications from other processes. A high-level systems programming language must be able to represent such non-deterministic, time-dependent behaviour.

Encapsulating Hardware

Although high-level languages are designed to be independent of hardware, systems programs must be able to control devices, respond to interrupts, schedule resources, etc. To make these aspects of the underlying hardware accessible to systems programs, it must be possible to represent them in the language.

Protection

Systems programming requires the ability to define components that are immune to interference from other erroneous or malicious programs. Such protection should be supported by the language.

Extensibility

The language should make it easy to create, in a modular fashion, new resources and services out of existing ones. This permits the expressive power of the language to be extended gracefully and incrementally.

Uniformity

There are significant advantages if a language uses uniform control and data structures, whether executing on one or several nodes. This makes it easier to reconfigure systems and to port programs to architectures of differing degrees of parallelism. (Unless a different architecture demands a different algorithm: an inherently sequential algorithm is unlikely to execute efficiently on a multiprocessor).

2.1.2 Efficiency

A systems programming language must be implemented on target hardware, by interpretation, compilation, etc. Its implementation must meet performance requirements. What these are depends on the application: a control program for a slow-moving industrial process may not need to execute as fast as an operating system, for example.

Efficiency is not determined solely by uniprocessor performance. In concurrent languages, communication and synchronization mechanisms must also be efficient, both on a single node and between nodes. A language that is to be used to program multiprocessors should not require centralized structures for its implementation, as these are likely to form a bottleneck.

2.1.3 Utility

A final, more subjective criterion determining the acceptability of a programming language is the benefits that are perceived to derive from its use. A high-level language that supports abstractions such as 'process' and 'communication' as language features is likely to make systems programming easier. Other factors that affect the utility of a language include its succinctness and the ease with which programs can be modified, reused, distributed across several nodes and statically or dynamically reconfigured.

The complexity and frequently critical nature of systems programming applications means that ease of verification is also important. Both a program's problem solving logic and its global behaviour (for example, real-time behaviour or termination) may need to be verified. Either the language should be itself sufficiently high-level to serve as its own specification language, or it should facilitate verification of programs from specifications written in some higher-level language.

2.2 Sequential Processes

The first class of languages considered represents attempts to incorporate concurrency into an *imperative* or *von Neumann* model of programming. Sequential imperative languages such as FORTRAN and PASCAL model computation as a sequence of read and write operations on shared data structures. The languages considered in this section extend this model to permit several sequential computations or **processes** to execute concurrently. These concurrent processes interact either through shared resources or by message passing. They may interact to exchange data (communicate) or to coordinate their execution (synchronize). Particular languages may be characterized by how they express concurrency and the communication and synchronization mechanisms they employ [Andrews and Schneider, 1983].

2.2.1 Expressing Concurrency

A number of language constructs for specifying concurrent execution of sequential processes have been proposed. These can be used to specify systems with a fixed or dynamic number of processes. Two such constructs are considered here: fork/join and parbegin.

2.2.1.1 Fork and Join

The **fork** statement [Conway, 1963] is used to represent process creation. A fork instruction creates a new process, which starts to execute a specified routine. The invoking process proceeds concurrently with the new process. The **join** statement permits the invoking process to synchronize with termination of the new process. Execution of this instruction delays the invoking process until a specified invoked process terminates.

The fork and join statements are a very general notation for specifying concurrent processes. They are widely used in the Unix operating system [Ritchie and Thompson, 1974]. However, their lack of structure can result in programs that are difficult to understand, particularly if they occur in loops or conditionals.

2.2.1.2 Parbegin

The **parbegin** statement [Dijkstra, 1968b] provides a more structured representation of parallel execution. A statement:

parbegin S_1; S_2; ...; S_n; parend;

denotes concurrent execution of the statements S_1, S_2, ..., S_n. Execution of this statement terminates only when all the S_i have terminated. The S_i can themselves be parbegin statements.

parbegin is less powerful than fork, as it can only be used to create a fixed number of processes. Also, it cannot represent such complex process dependencies: it is only possible to represent trees of processes, where each node waits for its offspring to terminate before terminating itself. However, a consequence of these limitations is that programs that use parbegin are easier to understand.

Languages incorporating parbegin include CSP [Hoare, 1978] and Occam [Inmos, 1984]. These are described below.

2.2.2 Communication using Shared Resources

Languages that use shared resources for interprocess communication may require synchronization mechanisms to control access to shared resources. Two types of synchronization can be distinguished. **Mutual exclusion** restricts access to a resource whilst a critical operation (such as reading and updating a shared variable) is performed. **Condition synchronization** delays access to a resource until some condition (such as 'buffer is full') is satisfied.

Both types of synchronization can be implemented explicitly using **busy-waiting**. Busy-waiting requires a process that wishes to perform some action to wait until a particular location has a specified value. However, busy-waiting is somewhat clumsy and furthermore wastes processor cycles. More abstract synchronization operations have therefore been defined. Two are described here: semaphores and monitors.

2.2.2.1 Semaphores

A semaphore [Dijkstra, 1968a] is an object on which 'set' and 'reset' operations (commonly called P and V) are defined. A process that attempts to set a semaphore that is already set is delayed. If a process attempts to reset a semaphore and processes are delayed waiting to set that semaphore, the semaphore is not reset but a delayed process is resumed. Semaphores are used to implement mutual exclusion as follows. A semaphore is associated with each resource to which access is to be controlled; this is initially reset. A process that wishes to access such a resource sets its semaphore, accesses the resource and then resets the semaphore.

Semaphores are higher-level and hence easier to use than busy-waiting. Furthermore, they may be supported by an operating system kernel, which can implement set and reset operations by moving processes to and from a ready queue — a queue of processes that can be executed — and suspension queues containing processes waiting to set different semaphores. This avoids wasting processor cycles.

However, semaphores do not enforce any structure on synchronization operations. It is thus rather easy to misuse them. For example, it is easy to miss out a set or reset operation or to apply such an operation to the wrong semaphore. Also, as set and reset operations may be scattered through code that is not concerned with synchronization, programs that use semaphores can be difficult to understand.

2.2.2.2 Monitors

Monitors were proposed by Hoare [1974] and others as a more structured way of controlling access to a shared resource. A monitor consists of a state and a number of procedures that define operations on that state. A monitor can thus be regarded as a module [Parnas, 1972] or abstract data type. The implementation of monitors ensures that processes executing procedures associated with a monitor are mutually excluded.

Queue variables [Brinch Hansen, 1975] may be defined within monitors to provide condition synchronization: that is, to delay processes that invoke monitor procedures until certain conditions are satisfied. Two operations are defined on these variables: **delay** delays the process which executes it until another process resumes it; **continue** resumes a process. The use of these primitives is illustrated in Section 2.2.2.3.

Monitors can be used in systems programming languages to encapsulate hardware resources. For example, an I/O device can be represented as a monitor on which read and write operations are defined. A call to this monitor returns when the I/O operation has completed. This hides the existence of hardware devices and interrupts from the high-level language programmer.

2.2.2.3 Concurrent PASCAL

Concurrent PASCAL was the first language to use monitors. Concurrent PASCAL [Brinch Hansen, 1975] and related languages have been used to implement operating systems for both uniprocessors and shared-memory multiprocessors [Brinch Hansen, 1976; Joseph *et al.*, 1984]. A Concurrent PASCAL system program creates a fixed number of concurrent processes which use monitors to access shared data structures. Monitors are also used as an abstraction for I/O devices and other hardware resources.

In Brinch Hansen's SOLO operating system, these abstractions are supported by a kernel of only 4K words of assembler.

```
type   outputprocess = process(buff: linebuffer)     % Part 1: define outputprocess.
       var item: char;                               %   Repeatedly ...
       while true do                                 %     compute and write chars
          compute(item); buff.write(item);           %     to buffer using monitor buff.

type   printerprocess = printer(buff: linebuffer)    % Part 2: define printerprocess.
       var text: line;                               %   Repeatedly ...
       while true do                                 %     read and print buffer
          buff.read(text); print(text);              %     using monitor buff.

type   linebuffer = monitor                          % Part 3: define monitor.
       var buffer: line; count: integer;             %   State variables.
          sender, receiver: queue;
       procedure write(item: char)                   %   Write character to buffer.
          if count = N then delay(sender);           %     delay writer if full.
          buffer[count] := item; count := count+1;   %     add item to buffer.
          if count = N then continue(receiver);      %     resume reader if full.
       procedure read(text: line)                    %   Read line from buffer.
          if count < N delay(receiver);              %     delay reader if not full.
          text := buffer; count := 0;                %     empty buffer.
          continue(sender);                          %     resume writer.

       begin count := 0 end;                         %   Initialization: empty buffer.

var output: outputprocess; printer: printerprocess;  % Part 4: create processes,
       buff: linebuffer;                             %     create monitor.

init output(buff), printer(buff);                    % Part 5: start execution.
```

Program 2.1 Printer buffer in Concurrent PASCAL.

Program 2.1 illustrates the use of the language. (For brevity, program blocks are delimited here by indentation rather than begin ... end pairs). This program defines (Parts 1-3) and creates (Part 4) concurrently executing output and printer processes and a monitor, linebuffer, which is used to buffer characters generated by output until a line is ready for printer. The output and printer processes are initiated using an init statement (Part 5). Concurrent PASCAL's init statement can be regarded as a form of parbegin. It is used to create static process structures: as instances of processes must

be explicitly declared, they cannot be created dynamically.

The monitor has as state variables a line buffer (buffer), a character count (count) and two queue variables (sender and receiver). It provides routines read and write which output and printer can use to access the buffer, plus initialization code which sets the character count to zero.

The write procedure adds a character to the buffer; read returns a full buffer. The state variables are used to synchronize the use of these procedures so that the buffer is neither written to when full nor read when not full. The state variable sender is used in write to delay the output process when the buffer is full and in read to resume it once the buffer has been emptied. The state variable receiver is used in read to delay the printer process when the buffer is not full and in write to resume it when it is full.

2.2.3 Communication using Message Passing

In languages based on message passing, processes communicate by sending and receiving messages rather than by accessing shared variables. Messages are sent using a **send**(<destination>, <value>) command and received using a **receive**(<source>, <variable>) command, where <destination> and <source> designate a destination and source. Languages based on message passing differ according to whether processes are named directly or indirectly, whether sources and destinations must be specified at compile-time or may be computed at run-time and whether communication is synchronous or asynchronous.

Direct naming requires that a process name be specified in send and receive commands. This is easy to implement but inflexible, as a process can only send to or receive from a named process, rather than *any* process, as is sometimes required. Indirect naming schemes introduce a level of indirection and hence permit many to one or many to many communication. The send and/or receive commands may designate a port (see Section 4.6) or other indirect construct.

Many languages (for example, CSP) require that sources and destinations be fixed at compile time. This is easy to implement, but precludes the programming of reconfigurable systems. Alternatively, processes may be permitted to compute values for sources and destinations at compile-time.

Synchronous communication means that both sender and receiver must be ready to communicate for communication to proceed. This can require two-phase communication protocols which ask for permission to send before sending data.

Asynchronous communication permits a sender to send a message that a receiver is not ready to receive. As the sender can then run arbitrarily ahead of the receiver, buffering is required in the implementation.

In languages based solely on message passing, there are no shared resources. One consequence of this is that a resource that may need to be accessed by more than one process must be represented as a process. Other processes send this process a message to access and modify the state of the resource. Such processes can be used to encapsulate hardware resources in a similar way to monitors.

2.2.3.1 CSP and Occam

Communicating Sequential Processes (CSP) [Hoare, 1978] is a programming notation based on synchronous message passing, guarded commands [Dijkstra, 1975] and a form of parbegin statement. Occam [Inmos, 1984] is a programming language derived from CSP. It is essentially a subset of CSP with a more readable syntax and extensions that address operational issues. It was designed as a systems programming language. In particular, it has been used as the machine language for the Transputer, a VLSI microprocessor that has been used to construct parallel computers. Occam is described here.

Occam programs combine three types of command — assignment, input (receive) and output (send) — in SEQ, ALT and PAR blocks. SEQ blocks represent sequential evaluation. ALT blocks express guarded commands. PAR blocks are a type of parbegin statement. Indentation is used to indicate the extent of blocks.

Assignment commands assign values to variables. They have the form variable := value. Input and output commands permit parallel processes to communicate using named, unidirectional communication links termed **channels**. They have the form channel ? variable and channel ! value, respectively. Message passing is synchronous: either sender or receiver delay until the other is ready to communicate.

A guarded command consists of two or more statements, each of which comprises a guard and a body. A guard is a conjunction of boolean expressions and input commands. A guard succeeds if its boolean expressions evaluate to true and no input command would be delayed. It fails if any boolean expression evaluates to false. Otherwise it delays. Evaluation of a guarded command evaluates the guards of each statement. If one or more guards succeeds, a non-deterministic choice is made of one of them and the commands in its body are executed.

Processes created within a PAR block communicate using channels. Predeclared

channels like 'keyboard' and 'screen' provide an interface to I/O devices. When executing on several nodes, Occam permits the programmer to specify where processes are to be executed.

```
PROC buffer(CHAN from_writer, from_printer, to_printer) =   % Buffer repeatedly
  VAR char, count, line [BYTE N]:                           %   processes writer
  SEQ                                                       %   or printer requests.
    count := 0
    WHILE TRUE
      ALT
        (count < N) & from_writer ? char                    % Buffer not full and
          SEQ                                               %       character received:
            line [BYTE count] := char                       %   Add to line buffer, &
            count := count + 1                              %       increment count.
        (count = N) & from_printer ? ANY                    % Buffer full and
          SEQ                                               %       line requested:
            to_printer ! line                               %   Send line to printer, &
            count := 0:                                     %       reset count.
PROC printer(CHAN to_printer, from_printer) =               % Printer repeatedly
  VAR line:                                                 %   requests and prints
  WHILE TRUE                                                %   lines.
    SEQ
      from_printer ! ANY
      to_printer ? line
      "print line":
PROC writer(CHAN from_writer) =                             % Writer repeatedly
  VAR char:                                                 %   generates and
  WHILE TRUE                                                %   sends characters.
    SEQ
      "compute char"
      from_writer ! char:
CHAN from_writer, from_printer, to_printer:                 % Declare channels.
PAR                                                         % Create processes.
  buffer(from_writer, from_printer, to_printer)
  writer(from_writer)
  printer(from_printer, to_printer)
```

Program 2.2 Printer buffer in Occam.

Program 2.2 implements the same buffer as Program 2.1. The buffer is represented here by a process, buffer. This maintains a line buffer and character count and has channel connections from an output process and to and from a printer process.

It accepts characters from output while the line buffer is not full and line requests from printer when the buffer *is* full. Having accepted a line request, it sends a line to printer and empties the buffer.

Occam has been widely used for programming parallel algorithms and parallel computers. An attractive feature of the language is its simplicity, which has permitted efficient implementation in hardware and the development of verification rules. However, the language can only implement static process and communication topologies and does not support recursion. This constrains the range of parallel algorithms that can be programmed in the language. Brinch Hansen [1987] has recently described a language that removes certain of these restrictions. It is not yet clear that similar verification rules apply.

2.2.3.2 Ada

Ada [DoD, 1982] is a language designed for use in process control applications. It provides a higher-level construct than `send` and `receive` for interprocess communication and synchronization: a form of remote procedure call termed the **rendezvous**. (Ada processes — termed **tasks** — can also communicate by shared variables).

A remote procedure call is a higher-level message passing construct that represents two communications: one to request a remote service and one to return a result. A process uses a callstatement to communicate a request to a named process. The request invokes a named procedure in the remote process; this invocation is (generally) implicit. This request is blocking, so the process making the request is delayed until a reply is received.

The rendezvous differs from an ordinary remote procedure call in that the receiving process can selectively receive requests using an accept statement. This permits a process to accept several types of request and to service requests when it desires. An accept statement waits for a matching request, processes the request and generates a reply.

2.2.4 Discussion

Two types of concurrent languages based on sequential processes have been identified: those that use shared resources for interprocess communication and synchronization and those that use message passing. Each has its advantages and

disadvantages. Shared resources introduce complex synchronization problems. These can be partially hidden using abstractions such as monitors, but still stretch the programmer's ability to visualize computational processes. Also, shared resource languages are difficult to implement efficiently on non-shared memory multiprocessors, as the overhead of maintaining shared mutable structures consistent is likely to be high. Message passing languages, on the other hand, tend to suffer from restrictive 'call-by-value' message passing: messages can only contain values, not references to data structures. Also, it can be difficult to implement message passing languages as efficiently as shared variable languages on shared memory machines.

Other points of interest include:

Encapsulation: both types of language can encapsulate hardware resources quite elegantly, as monitors and processes respectively. Both monitors and processes provide, or can be used to provide, some protection against illicit access to resources.

Uniformity: languages based on sequential processes are not uniform in the sense that programs have distinct sequential and parallel components. In the case of message-passing languages, these components use quite different mechanisms to perform computation: assignment and message passing, respectively. This lack of uniformity can make programs harder to understand and reconfigure. For example, a program may execute efficiently on N nodes if it has been specified in terms of N processes. However, it must be completely rewritten to exploit 2.N nodes.

Simplicity vs expressiveness: the languages considered trade conceptual simplicity for expressive power. Simple languages (such as Occam) are easy to implement and verify but of limited expressive power. Complex languages (such as Ada) may be more expressive but have a complex semantics that hinders verification.

2.3 Actors

Actors are an abstract notation for specifying concurrent computation [Hewitt, 1977; Agha, 1986]. In contrast to the languages described in the previous section, which are based on a sequential model of computation and in which parallel processing is specified by additional constructs, the actor formalism is intrinsically concurrent.

An actor system consists of a number of actors. An **actor** is an agent that is defined by an **address** and a **behaviour**. Its behaviour determines how it processes messages sent to its address. In general, an actor maps a message into a finite set of messages sent to other actors, a new behaviour and a finite set of new actors. An actor can

communicate with any actor for which it knows the address; such actors are known as its **acquaintances**. Addresses can be passed between actors in messages. Messages are assumed to be forwarded by an underlying mail system that guarantees only that each message sent will eventually be delivered exactly once.

The actor formalism is thus characterized by dynamic processes, dynamic communication topology, asynchronous communication and global reference (but not a global address space).

```
def buffer(n, index, previous, link, reader) [element]   % Receive an element
    if index < n then                                    % Buffer not full?
        let P = new 1_buffer(previous, link)             %   Create new actor.
        become buffer(n, n+1, element, P, reader)
    if index = n then                                    % Buffer full?
        let P1 = new 1_buffer(previous, link)            %   Create new actors:
        let P2 = new 1_buffer(element, P1)               %    prev. and last element.
        send P2 to reader                                %   Pass to reader.
        become buffer(n, 0, NIL, SINK, reader).          %   Start again.
```

Program 2.3 Buffer actor.

Program 2.3 illustrates the formalism. It implements a simple buffer in a language similar to the language SAL defined by Agha [1986]. This buffer accepts messages until it has collected n of them and then communicates the n elements it has collected to another actor. It differs from the buffers implemented in Programs 2.1 and 2.2 in that it does not constrain the production of elements: any actor that knows the buffer's address can generate and transmit elements to it.

The buffer definition consists of two guarded statements that specify how to process communications. Note the use in the program of the **new** command, which creates a new actor and returns its mail address, the **send** command, which generates a communication, and the **become** command, which specifies a new behaviour.

The buffer accepts messages representing elements to be buffered and creates new actors (1_buffer: not defined), each of which possesses the address of an element and of the previous 1_buffer actor. These new actors thus represent a 'list' of elements. When the buffer has received n of these messages, it passes the address of the head of this 'list' to the actor it knows as reader. This may then send messages to the 1_buffer actors to access the buffered elements.

A buffer may be created as follows:

```
let b = new buffer(n, 0, NIL, SINK, reader)
```

where n is the size of the buffer, 0 is the number of elements collected so far, NIL is some predefined value used to signify the end of the buffer, SINK is the address of some 'dummy' actor and reader is the address of the actor to which communications are to be sent when the buffer becomes 'full'.

The actor formalism is a powerful tool for representing concurrent systems. Attractive features include its simplicity and uniformity — both control and data structures are represented by actors — and its inherent concurrency. However, actors are a very low-level formalism, a sort of machine language. Programming languages based on this formalism must provide higher-level abstractions.

Interestingly, the parallel logic languages share many features with actors. The parallel logic languages are distinguished by their declarative foundations, recursive data structures, simple syntax and built-in unification mechanism. Kahn and Miller [1988] emphasize the similarities between the two formalisms.

2.4 Functional Languages

Functional programming languages are based on the lambda calculus, a mathematical theory of functions [Church, 1941]. Functional languages and the benefits they can bring to programming are described in [Backus, 1978] and [Henderson, 1980]. The first and best known functional language is LISP [McCarthy *et al.*, 1965], though most LISP dialects incorporate imperative features. Modern functional languages include HOPE [Burstall *et al.*, 1980] and SASL [Turner, 1981].

The usual model of computation employed in functional languages is extremely simple. A functional program defines a set of rewrite rules which are used to evaluate expressions containing rewritable functions. A rewritable function is evaluated by finding a rewrite rule with a left hand side that matches the function and replacing that function in the expression by the right hand side of that rewrite rule. This process continues until an expression that contains only constructor functions is produced. Constructor functions represent structured data such as lists.

For example, consider the following functional program. This can be both read as a definition of list membership and executed to find the value associated with a particular key in a list. Strings beginning with capital letters denote variables.

```
lookup(Key, [ ]) = missing
lookup(Key, [ [Key1,Value] | Data] ) =
        if Key = Key1
        then Value
        else lookup(Key, Data).
```

(In this program and throughout this thesis, a special syntax is used for the type of structured term known as **lists**. The notation [H | T] is used to denote the list with head H and tail T. A nested list term [X | [Y | Z]]may be abbreviated as [X, Y | Z]or if Z is the empty list — denoted by the constant [ ] — as [X, Y]).

A function call lookup(john, [[mary, 28], [john, 26]]) returns the value 26. A function call lookup(john, [[mary, 28]]) returns the value missing.

The basic rewrite model is usually augmented with **lazy evaluation** [Henderson and Morris, 1976; Friedman and Wise, 1976]. This is a demand-driven evaluation strategy which only evaluates functions when their values are required. This does not alter the declarative semantics of functional programs, but makes problems that deal with potentially infinite data structures tractable. Kahn and MacQueen [1977] generalize the functional model to allow both **restricted** concurrency (where multiple arguments to a function are evaluated concurrently) and **stream** concurrency (concurrent evaluation of a function and its argument).

The rest of this section first describes work on systems programming in pure functional languages and then looks at various attempts to overcome perceived limitations in pure functional programming.

2.4.1 Systems Programming in Functional Languages

Systems programming and mathematical functions may appear to have little in common. Although a number of LISP machines use LISP as a machine language, these systems rely on side-effecting primitives and must be regarded as imperative systems with a functional syntax. Several researchers have however explored the use of pure functional languages for systems programming. The basic insight is that operating systems can be viewed as functions that map external events to side-effects in the external world. Henderson [1982] and Jones [1984] show how simple operating systems can be defined as functions that map keyboard input to screen output. These programs represent an operating system as a set of recursive functions that communicate by incrementally constructing and consuming potentially infinite data

structures (usually lists) called **streams**. Lazy evaluation ensures that data is only demanded from the user as it is required to calculate an output value. Stream concurrency permits concurrent evaluation of the various functions defining the system. The underlying implementation is assumed to provide interfaces to the outside world by generating the input stream and displaying the output stream.

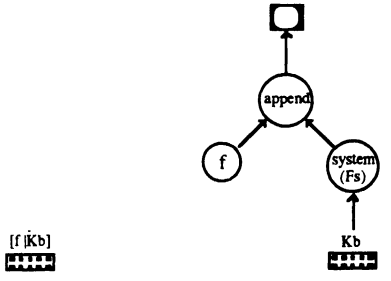
Jones describes a number of purely functional operating systems constructed using these techniques. For example, Program 2.4 defines a very simple operating system that repeatedly executes functions named by the user if they can be found in a supplied filestore Fs.

```
system([Function | Kb], Fs) =                                        % Receive input.
    if lookup(Fs, Function ) = missing                               % Defined ?
    then [ error | system( Kb, Fs) ]                                 %  No: ERROR.
    else append( execute( lookup( Fs, Function ) ), system( Kb, Fs) ) %  Yes: execute.
```

Program 2.4 Simple functional operating system.

Program 2.4 generates an output stream consisting of either function output or the constant error if Fs does not contain the definition of a named function. Matching its first argument with the data structure [El | Kb] is assumed to return the next term El input by the user. Kb is the remainder of the input stream. The implementation is assumed to display system's output at the user's terminal.

Figure 2.1 represents the execution of this program. An input naming a function f results in evaluation of functions f, append and system, if f is defined in the structure Fs representing the filestore.



1. Input 'f' received. 2(a) f defined in Fs. 2(b) f not defined.

Figure 2.1 Execution of simple functional operating system.

Functional programs are referentially transparent: that is, evaluation of a function always gives the same result, in whatever context it is evaluated. A consequence of this is that functions are history-insensitive. This example shows how internal state (in this case, a filestore) can be represented as an argument to a function whose iterative evaluation defines a system. Changing state can also be implemented in this way: for example, system can be redefined to accept messages containing new functions to add to the filestore.

2.4.2 Non-determinism

Operating systems must in general process input as it becomes available from several independent sources. In functional terms, this means that a function must sometimes be able to return only one of a set of values being generated by concurrently evaluating arguments. For example, a function may have two arguments, one of which evaluates to a stream of keyboard input and the other to a stream of interrupts. Either keyboard input or interrupts must be processed as they become available. Pure functions cannot describe this time-dependent or **non-deterministic** behaviour.

2.4.2.1 Non-deterministic Primitives

A number of solutions to this problem have been proposed. The simplest is to extend the functional language with a non-deterministic primitive. This can either be a merge primitive (as proposed by Henderson [1982] and Abramsky [1982]) or some lower-level operator that can be used to program merge and other operations, such as frons [Friedman and Wise, 1980] or a non-deterministic choice operator [Jones, 1984]. All of these approaches provide (or can be used to define) a 'non-deterministic' merge which, when applied to two streams a and b generates a stream c that is a 'non-deterministic' interleaving of the elements on a and b. The stream c contains all the elements of a and b, with the ordering of elements in a preserved, the ordering of elements in b preserved but with the elements of a and b interleaved in some arbitrary fashion. The use of the merge operator to combine output from two concurrently evaluating functions is illustrated in Section 2.4.5.

Merge is non-deterministic in the sense that the interleaving of elements in the output stream is not determined by any functional description. To be useful, merge must also be fair: it should not ignore one stream indefinitely. Any operator used to implement merge therefore needs to provide some analogue of the operational concept

of **fairness** [Friedman and Wise, 1979].

Some form of non-determinism appears essential if functional languages are to be used for systems programming. Unfortunately, non-determinism compromises the mathematical purity of functional programs and invalidates certain transformation and proof techniques. For example, the expression:

```
if X = X then yes else no
   where X = E
```

always gives the result yes for any expression *E*. On the other hand, the apparently equivalent expression:

```
if E = E then yes else no
```

may give result no if *E* contains non-deterministic components. In this case, *E* is evaluated twice; a non-deterministic choice may cause it to compute different values.

2.4.2.2 Explicit Treatment of Time

The time-dependent nature of many non-deterministic choices has motivated proposals that time-stamps be associated with data items [Broy, 1983; Harrison, 1987]. Alternatively, an implicit time parameter may be introduced, as in the LUCID family of languages [Ashcroft and Wadge, 1976]. The value of a variable is then regarded as a continuous sequence of values, each value applying at a different time. This permits variables to change value over time, as in imperative languages. Yet programs retain a declarative semantics.

These approaches permit a purely functional treatment of asynchronous events. It is not clear however that they can be efficiently implemented. Also, the explicit treatment of time appears to result in rather clumsy programs. Thus, though LUCID has been used as a specification language for distributed and real time systems [Glasgow and McCewan, 1987], it has not proved feasible to implement such systems in the language.

2.4.2.3 Stoye's Scheme

Stoye [1984] proposes another scheme which avoids the need for an explicit non-deterministic operator. A number of functional expressions (processes) are evaluated in parallel, each with an input and output stream. The underlying system is assumed to provide a router which takes addressed messages off the output streams and distributes

them to the appropriate input streams. This has the advantages of restricting non-determinism to one place (the router) and encouraging efficient implementation (the router can be optimized). However, it associates the overhead of decoding an address with each message. Also, as it constrains the way non-deterministic constructs are used, it may reduce the utility of these constructs.

Stoye's scheme avoids another potential problem with functional languages. Communication channels in functional languages (streams) are defined by function applications and are thus static. Programming complex systems in functional languages often requires so many streams that the resulting 'spaghetti' is hard to understand. This effect is exacerbated by the unidirectional nature of functional communication, which means that most communication links require both a request and a reply stream. In Stoye's scheme, each entity has a single input and output stream. Also, dynamic connections between processes can be established: a process can include a 'return address' in a message to a resource such as a filestore.

Arvind and Brock [1982] describe another solution to the problem of static communication channels. They introduce a construct called a **manager** into a functional language. This encapsulates a shared resource and handles the routing of replies to processes making requests to the resource.

2.4.3 Synchronization

Systems programming often requires the synchronization of events. For example, a prompt should only be displayed when user computation has ceased. Time-critical events such as interrupts should be processed before other computation. In functional terms, synchronization requires that function evaluations be sequenced in a way that is independent of data dependencies. However, unless assumptions are made as to the operational behaviour of the language implementation (which is not desirable) then the order in which functions are evaluated is not determined. Functional languages are not therefore naturally suited to the description of temporal constraints and synchronization.

Darlington and While [1986] propose that temporal requirements be stated explicitly in a temporal metalanguage which permits the specification of total and partial orderings on function rewrites. A functional program plus its temporal description is transformed into an imperative program that displays the desired behaviour. This approach has the advantage of providing a clear specification of the temporal component of a program. Also, the programmer only needs to specify the minimal ordering required. However,

its implementation is based on shared mutable data structures; this introduces imperative features, which could cause difficulties on certain architectures.

2.4.4 Unification

Functional languages use matching to determine whether a rewrite rule can be used to rewrite an expression. Matching is a unidirectional operation: functions consume data structures constructed by their arguments. Logic programming languages (see Section 2.5) use **unification** rather than matching during reduction. This permits bidirectional dataflow. It means that a producer can construct data structures with unspecified (variable) components which are subsequently given a value by the consumer of the data. This is known as the **logical variable** and has proved to be a powerful feature of logic programming.

An alternative operational semantics for functional programs has been proposed that replaces matching with unification. This is **narrowing** [Reddy, 1985a]. Narrowing adds to the expressive power of functional programs. However, languages based on narrowing lose some of the simplicity of functional programming and inherit the implementation problems of the logical variable, particularly when parallel execution is considered. As the evaluation of an expression computes a set of narrowings, parallel evaluation of two expressions with shared variables may generate conflicting bindings for variables, which must be resolved. Reddy [1985a] suggests that lazy narrowing can help reduce conflict when evaluating expressions in parallel and argues that function application provides a natural notation for sequencing computations when sequential evaluation is desired. It remains to be seen, however, whether efficient implementations of narrowing are possible.

Languages based on narrowing include Qute [Sato and Sakurai, 1984] and Eqlog [Goguen and Meseguer, 1984]. Eqlog introduces the logical variable into an equational language and uses narrowing to find general solutions to equations. It is more a logic language than a functional language: it does not, for example, support lazy evaluation or higher-order functions. The language is undoubtedly powerful and efficient implementation techniques are apparently being developed. It is not yet clear however whether these can be extended to support concurrency and non-determinism, without which the language will not be useful for systems programming.

Another attempt to introduce the logic variable into functional programming is HOPE with unification [Darlington *et al.*, 1985]. This introduces **set abstractions**

which allow logical variables within set expressions on the right hand side of function-defining equations. This provides a restricted form of logical variable within the usual functional computational model.

2.4.5 Parallel Execution

Functional languages are well-suited to parallel evaluation. Their referential transparency means that functions can be evaluated in any order or in parallel and still yield the same result. Various novel computer architectures have been proposed that attempt to exploit the potentially massive parallelism expressed by functional programs [Treleaven *et al.*, 1982; Vegdahl, 1984]. The ALICE machine, for example, implements a model of computation in which multiple agents repeatedly select rewritable functions from a global pool, apply a rewrite rule and place the resulting expression back in the pool [Darlington and Reeve, 1981].

Functional languages can also be implemented on more conventional architectures such as non-shared memory multiprocessors. Problems then arise of partitioning functional programs so as to maximize useful computation and minimize communication between nodes [Keller and Lin, 1984]. This may be achieved by compile-time analysis, by run-time support for dynamic load-balancing or by explicit mapping annotations which indicate on which node expressions are to be evaluated [Hudak, 1986]. For example, the notation: f on p may be used to indicate the evaluation of function f on node p. Notations such as f on left(self) can be used to express relative mappings on topologies that permit a notion of left and right. self evaluates to the identifier of the node on which it is evaluated.

As Henderson [1982] points out, functional operating systems such as Program 2.4 map naturally to parallel machines. Each function in a network of concurrently executing functions may be located on a different node. Stream communication between functions corresponds to internode communication. For example, using mapping notations, Program 2.4 may be reexpressed as follows:

```
system([Function | Kb], Fs) =
        if lookup(Fs, Function ) = missing
        then [ error | system( Kb, Fs) ]
        else merge( execute( lookup( Fs, Function ) ), system( Kb,Fs) on left(self) )
```

This specifies that upon receiving a valid input, the new function shall start executing on the current processor whilst the operating system (system) continues

executing on an adjacent node. Note the use of merge rather than append as in Program 2.4; this intermingles rather than concatenates the output of this and subsequent function evaluations. Several function evaluations may therefore proceed concurrently, each executing on a different node.

2.4.6 Discussion

The simple syntax and semantics of pure functional languages facilitate transformation and analysis. Their lack of side-effects and implicit parallelism permit efficient parallel implementation. Another advantage of the formalism is its uniformity.

Evaluation strategies incorporating lazy evaluation and stream concurrency permit the implementation of simple operating system structures as recursively defined functions. However, difficulties are encountered when pure functional languages are used for more complex systems programming tasks. Important aspects of the operational behaviour of computer systems cannot be defined in terms of function rewrites. These problems have motivated extensions to functional languages. These extensions are effective, but the resulting languages lose some of the simplicity of pure functions. Interestingly, extensions such as non-determinism and unification produce languages that are quite similar to the parallel logic programming languages (see Chapter 3).

2.5 Logic Languages

Logic programming [Kowalski, 1979], though based on a completely different abstract formalism — the restricted form of first order logic known as Horn clauses — has many similarities to functional programming. The relationship between functional and logic languages has been discussed by Kowalski [1983] and more recently by Darlington *et al.* [1985] and Reddy [1985b].

Logic programs are sets of logic clauses. A logic clause has the form:

$$A_0 \leftarrow A_1, \ldots, A_n.$$

where the A's are goals. It can be read declaratively as a logical sentence:

A_0 is true if A_1 and ... and A_n are true.

It can also be read as a procedure for proving A_0:

To prove A_0, prove A_1 and ... and prove A_n.

The following example, which implements the quicksort sorting algorithm, illustrates the formalism.

```
sort([E |L], S) ← partition(E, L, L1, L2), sort(L1, S1), sort(L2, S2), append(S1, [E |S2],S).
sort([ ], [ ]).

partition(E, [N |L], [N |L1], L2) ← N < E, partition(E, L, L1, L2).
partition(E, [N |L], L1, [N |L2]) ← N ≥ E, partition(E, L, L1, L2).
partition(E, [ ], [ ], [ ]).

append([E |X], Y, [E |Z]) ← append(X, Y, Z).
append([ ], Y, Y).
```

This program can be read as a set of logical sentences defining the sort, partition and append relations. Variables in a clause (denoted here by capital letters) are implicitly universally quantified. The first clause can be read declaratively as:

For all E, L, S, L1, L2, S1, S2: sort([E |L], S) if
partition(E, L, L1, L2) and sort(L1, S1) and sort(L2, S2) and append(S1, [E |S2],S)

or, in English:

"Sorting the list [E | L] gives the list S *if* partitioning L about E gives lists L1 and L2 *and* sorting L1 gives S1 *and* sorting L2 gives S2 *and* appending [E |S2] to S1 gives S".

The second clause reads: sorting the empty list gives the empty list.

The **logical reading** of a relation R in a program P is the relation that it describes. A term T_1, ..., T_k is a member (or **solution**) of a k-ary relation R if R(T_1,...,T_k) is a logical consequence of the program P. The logical reading of sort(L,S) is: S is the sorted version of L. sort([1,4,2,3],[1,2,3,4]) is a member of this relation. partition(E,L,L1,L2) reads: L is a list of items, L1 is the sublist of L comprising items less than E and L2 is the remainder of L. append(X,Y,Z): Z is a list comprising all the elements of X followed by Y.

A logic program is invoked by a query: a conjunction of goals. Variables in a query are implicitly universally quantified. The query:

```
?- qsort([1, 4, 2, 3], X).
```

is read as 'Does there exist an X such that qsort([1, 4, 2, 3], X) ?'.

Computation in logic programming languages is controlled deduction: an attempt to prove a query using program clauses. The resolution-based evaluation strategies (see Section 2.5.1) used in most logic programming languages are constructive: that is, in order to prove that there exists a solution to a goal, they compute values for its variables. These values, which define a member of the relation invoked by the goal, constitute the output of the computation.

This means that logic clauses can be read as procedures for computing members of the relations that they define. For example, sort may be read as a procedure for sorting lists: To sort a list [E | L], partition L about E to get L1 and L2, sort L1 to get S1, sort L2 to get S2 and append [E | S2] to S1 to get the sorted list. To sort the empty list, return the empty list.

The query given above invokes this sort procedure. Its solution is X = [1,2,3,4]. Whether or not a particular logic programming language can compute this solution depends on its evaluation strategy. The set of solutions that a logic programming language is capable of computing is not necessarily equivalent to the set of solutions implied by the declarative reading of a program: in most logic programming languages, it is a subset.

2.5.1 Resolution and Unification

Most logic programming languages use an evaluation strategy based on Robinson's resolution principle and unification algorithm [Robinson, 1965; Kowalski, 1974]. Resolution attempts to reduce an initial set of goals (a query) to an empty set by a series of resolution steps. A resolution step consists of the following actions:

1. Select a goal G in the query Q.
2. Select a clause C from the program with the same relation name as G and rename variables so that all variables in C are new, that is do not occur in Q.
3. Unify the goal G with the head of clause C, with most general unifier θ
4. Replace G in Q with the body of C.
5. Apply θ to Q to generate a new query Q', to be used in the next resolution step.

A resolution step may fail if any of the actions 2-5 are not possible. Depending on the **control strategy** that is being used to direct resolution, this failure may be fatal or may lead to the repetition of a previous resolution step using an alternative clause. A

control strategy determines in what order resolution steps are performed and which goal and clause are selected at each resolution step. Different control strategies correspond to different logic programming languages. Some logic programming languages, such as Parlog (see Chapter 3), allow the programmer to guide the application of the control strategy using a separate **control language**.

If resolution succeeds in reducing an original query to the empty query by a series of resolution steps, it has succeeded in proving the query. Resolution is constructive: it may give values to variables in the original query. These bindings may be regarded as the output of the query.

Variables are instantiated (given values) by unification. Two terms *G* and *H* are said to **unify** if there exists a substitution θ of values for variables in *G* and *H* which, when applied to the two terms, makes the terms syntactically identical. The unifying substitution should be the most general unifier of *G* and *H*: that is, a unifying substitution that is not an instance of any other unifying substitution, except by variable to variable substitution.

A substitution is a set of (*V*/*T*) pairs, each of which indicates that variable *V* is to be replaced with term *T*. Thus the terms:

p(a,X) and p(Y,g(Z))

unify with most general unifier:

{(X/g(Z)), (Y/a)}.

generating the term:

p(a,g(Z))

Unification is the fundamental data manipulation operation in logic programming. It bidirectional nature makes it extremely powerful, subsuming a wide range of conventional data manipulation operations such as pattern matching, structure access, parameter passing and data construction (such as *cons*). Its power is also apparent in the many applications of the logical variable, some of which are described in the chapters that follow.

2.5.2 Prolog

Resolution can be viewed as a generalization of function rewriting that permits a non-deterministic choice of rewrite rule and uses unification rather than pattern matching. It may thus appear that logic programming languages can be used for systems programming in a similar way to functional languages, with non-determinism and the logical variable providing added expressive power.

The best-known logic programming language, Prolog [Roussel, 1975], performs a single resolution step at a time. Goals are reduced left to right and clauses are tried in textual order. If a resolution step fails, **backtracking** causes a previous resolution step to be repeated using an alternative clause. Only if no alternative clause exists in any previous step does resolution fail.

Prolog's evaluation strategy is thus characterized by sequential evaluation and a built-in search mechanism. Despite apparent similarities between a resolution step and a functional rewrite, Prolog's computational model is very different from the function rewrite model.

Prolog's search mechanism accounts for much of the language's expressive power in applications such as natural language parsing. Prolog's sequential evaluation strategy permits efficient implementation on sequential machines, as stacks can be used to represent the state of an evaluation [Warren *et al.*, 1977]. However, neither sequential evaluation nor search are necessarily appropriate in systems programming applications.

2.5.3 Don't-Know and Don't-Care Non-determinism

Several clauses in a logic relation may match with a particular goal. It is not in general possible to know which clause must be evaluated to compute a solution. This underspecification is referred to as **don't-know non-determinism** [Kowalski, 1979]. Prolog deals with it by searching through all clauses until a solution is found.

Don't-know non-determinism means that a solution constructed by a logic program computation cannot be trusted until the computation terminates. Subsequent evaluation may discard any such 'provisional solution' computed prior to termination and use an alternative clause to compute a completely different solution. This prevents the definition of operating systems as logic programs that compute relations over 'infinite' input and output streams.

In contrast, rewrite rules in pure functional languages are deterministic. Any value

calculated by a functional program is hence definite and can, for example, be output with confidence. This permits operating systems to be defined as functions that consume and generate infinite streams.

Provisional values computed by a logic program can be output if a committed choice of clause is made when several can be used to reduce a goal. Alternative clauses are discarded. This **don't-care non-determinism** (the programmer 'doesn't-care' how a solution is obtained) means that although subsequent evaluation may lead to failure, values computed so far cannot be retracted and can hence be trusted. In Prolog, don't-care non-determinism can be realized using the rather inelegant 'cut' primitive, which permits the programmer to abandon alternative solutions. Don't-care non-determinism is introduced in a more elegant fashion in the parallel logic languages (see Chapter 3).

2.5.4 Sequential Evaluation

Prolog cannot describe concurrent activity because of its sequential evaluation strategy. This problem can be overcome within the context of Prolog by introducing **coroutining** of goal evaluations as in IC-Prolog [Clark and McCabe, 1979] and NU-Prolog [Thom and Zobel, 1986]. In coroutining systems, the sequential evaluation strategy is relaxed. By default, goals are evaluated from left to right. However, if a goal nominated as a **consumer** of a particular value is called with that value unspecified, it is **suspended** and evaluation proceeds with other goals. Evaluation of a suspended goal is resumed when the value it is to consume is computed.

```
producer(X, Xs) when X
producer(Y, [X |Xs]) ← evaluate(Y), producer(X, Xs).

consumer(X) when X
consumer([X |Xs]) ← compute(X), consumer(Xs).

?- producer(X, Xs), consumer([X |Xs]).
```

Program 2.5 Coroutining in NU-Prolog.

Coroutining permits the programming of elegant search algorithms in which solutions are incrementally generated and tested [Clark and McCabe, 1979]. In systems programming applications, it provides the ability to define networks of goals

with directional dataflow. Consider the NU-Prolog program and query listed in Program 2.5. A when declaration associated with a procedure nominates that procedure as a consumer of a particular argument.

When the query is evaluated, the producer goal (nominated as a consumer of its first argument by its when declaration), initially suspends as its first argument X is not available. consumer is then evaluated. This calls compute to determine a value for X. This resumes producer, which 'evaluates' this value and generates a request for another value. This process proceeds indefinitely.

The query can be viewed as defining a network of two goals, producer and consumer. producer generates a stream of requests for values (Xs). consumer generates values in response to these requests. Interestingly, this single stream is used for bidirectional communication. This gives an indication of the power of the logical variable. Consider, however, what happens if a call to evaluate fails. Backtracking must occur in both producer and consumer to 'compute' a new value. The underlying computational model is still complex and inappropriate for systems programming.

2.5.5 State Change

Logic programming models computation as controlled deduction from a fixed set of logical axioms. This model does not naturally provide the notion of state and state change necessary if systems are to be history-sensitive.

Many Prolog implementations provide primitives such as assert and retract that can be used to write self-modifying programs. However, these have no logical semantics. Furthermore, by reintroducing side-effects, they make parallel implementation difficult. A declarative treatment of state change can be achieved in logic programming in a number of ways. For example, the event calculus [Kowalski and Sergot, 1986] models state change as a monotonic assimilation of information. Alternatively, logic languages can be extended so that programs can be represented as language objects [Bowen and Kowalski, 1982; Bowen, 1985]. It is then possible to write metaprograms: programs that define relations over terms interpreted as representing other programs. Temporal logics permit explicit description of state change (see Section 2.5.7). State can also be implemented as in functional languages (Section 2.4.1) as arguments to recursive procedures defining non-terminating systems. This avoids the need for side-effects but is not a declarative treatment of state change.

2.5.6 Parallel Execution

Logic programs, like functional programs, frequently express considerable implicit parallelism. Two types of parallelism may be distinguished in logic programs. **And-parallelism** is the parallel execution of two or more goals in a conjunction. **Or-parallelism** is the parallel evaluation of a goal using two or more clauses defining a procedure. The two types of parallelism can be exploited independently or together. Each has distinct implementation problems.

And-parallel evaluation leads to the problem that concurrently executing goals may generate conflicting bindings for shared variables. One of the goals must then be forced to backtrack and generate an alternative solution. This can be complex on a parallel machine. A proposed solution to this problem is to only evaluate goals in parallel when it can be determined at compile-time or run-time that they do not share variables [Conery and Kibler, 1981; DeGroot, 1984; Hermenigildo, 1986]. Alternatively, one goal can be specified as the producer of bindings for a shared variable and other goals as consumers of these bindings. This **stream and-parallelism** was investigated by van Emden and de Lucena [1982]. It is the most important form of parallelism in the parallel logic programming languages [Clark and Gregory, 1981].

The principal problem associated with or-parallel evaluation is maintaining alternative values for variables generated by different clause evaluations. A number of schemes for maintaining this information have been proposed [Crammond, 1985; Warren, 1987].

Or-parallel evaluation is best-suited to problems with large amounts of search: that is, don't-know non-determinism. While search can be required in systems programming (for example: for computing optimal resource allocations), in general the systems programmer wishes to apply an efficient algorithm to compute a desired solution, and does not require the implementation to perform a potentially inefficient search. Or-parallelism is thus likely to be less important than and-parallelism in systems programming applications. Experimental results that support this hypothesis are presented in Chapter 5.

2.5.7 Other Logics

Programming languages based on logics other than Horn clause logic and using proof procedures other than resolution have been proposed.

Temporal logic [Pnueli, 1986] has been used as a specification language for concurrent systems. However, efficient implementation techniques that would permit its use as a programming language have yet to be developed.

Temporal logic languages such as Tempura [Moszkowski, 1984] and Tokio [Aoyagi *et al.*, 1985], both based on a type of temporal logic termed Interval Temporal Logic, permit variables to take different values at different times by introducing notions of time and history. This permits explicit description of state change. Tempura is an essentially procedural language, however, and supports neither non-determinism nor unification. Tokio, which does support these features, is less sophisticated in other respects. A Prolog-style evaluation strategy makes it possible to "backtrack into the past", which means that the language's utility seems to be restricted to simulation.

Periera and Nasr's Delta-Prolog [1984] is based on a form of logic termed Distributed Logic [Monteiro, 1984]. It provides language primitives that can be used to create concurrent processes. These processes can synchronize and communicate using explicitly specified events. These mechanisms are similar to those found in CSP (see Section 2.2.3.1).

Alternative logics can permit a more elegant treatment of certain problems. However, it has yet to be shown that any of them is as widely applicable or as efficiently implementable as the logic programming languages.

CHAPTER 3

The Parallel Logic Programming Language Parlog

Parlog is both a logic programming language and a process-oriented language. Parlog programs are sets of logical axioms which have a declarative reading. They have an alternative behavioral reading in which they define systems of processes. These execute concurrently, communicate through shared logical variables and synchronize using dataflow constraints.

Parlog was designed by Clark and Gregory [1986]. Gregory [1987] provides an excellent account of the language. For tutorial introductions to parallel logic languages and their applications see [Shapiro, 1986; Ringwood, 1988]. Differences between Parlog and other parallel logic languages are noted in Section 8.2.

This chapter first describes Parlog's syntax and operational semantics. Section 3.3 introduces important Parlog programming techniques. Section 3.4 describes an important component of the language: a primitive that enables Parlog programs to monitor and control the execution of other programs. In Section 3.5, Parlog's declarative characteristics are considered.

3.1 Syntax

A Parlog program is a finite set of **guarded clauses** and **mode declarations**. A guarded clause has the form:

$$H \leftarrow G_1, \ldots, G_m : B_1, \ldots, B_n. \qquad m,n \geq 0$$

H is called the clause's **head**, $G_1, \ldots, G_m$ its **guard** and $B_1, \ldots, B_n$ its **body**. The ':' is a **commit operator**. If the guard is empty ($m = 0$), the commit operator is omitted.

The head H and each of the $G_1, \ldots, G_m$ and $B_1, \ldots, B_n$ are **goals**. (Goals are also referred to as **processes**). A goal has the form $R(T_1, \ldots, T_k)$, $k \geq 0$, where R is the **relation name** and the **arguments** $T_1, \ldots, T_k$ are terms. If $k = 0$, the brackets are omitted. R is said to have **arity** k. A relation with name R and arity k is sometimes denoted R/k.

Clauses with the same relation name and arity are grouped together into **procedures**.

A **mode declaration** is associated with a Parlog procedure. For a relation R of arity k, this has the form:

mode $R(m_1, ..., m_k)$. where each m_i is either '?' or '↑'

A '?' denotes **input** mode and a '↑' **output** mode. The modes m_i may be prefixed with argument names. These are comments of no semantic significance.

A Parlog program is invoked by a **query**. A query is a conjunction of goals:

?- $Q_1, ..., Q_p$ $p \geq 1$

A **term** is a variable, constant or structured term. A **variable** is denoted by an alphanumeric sequence beginning with a capital letter or the underscore character ('_'), such as X, Name or _1. The underscore character may also be used on its own to denote a unique, unnamed (*anonymous*) variable. A **constant** is an **integer** such as 0, 316 and -19 or a string. A **string** is any character sequence; if it could be confused with other types of term, it may be enclosed in single quotes. Thus: peter and 'Enter name: ' and 'CALL'.

A **structured term** may be written using **tuple** notation:

$\{T_1, ..., T_j\}$, $j \geq 1$

where the T_1, ..., T_j are terms. If T_1 is a string F, it may also be written as $F(T_2, ..., T_j)$, and F is termed the **function name**.

$\{F, T_2, ..., T_j\}$ is equivalent to $F(T_1, ..., T_j)$

As noted in Section 2.4, a special syntax is used for the type of structured term known as **lists**. The notation [H | T] is used to denote the list with head H and tail T. A nested list term [X | [Y | Z]]may be abbreviated as [X, Y | Z]or, if Z is the empty list or **nil** (denoted by the constant [ ]), as [X, Y].

Goals in a clause or query may be separated by the operators ',' or '& '. These are Parlog's **parallel** and **sequential conjunction** operators, respectively. Similarly, clauses in a procedure may be separated by the operators '.' or ';'. These are Parlog's **parallel** and **sequential clause search** operators. '.' is also used to terminate a procedure.

Infix notation may be used for goals involving binary relations. For example, the goal =(X,Y) may be written X = Y. Also, programs may include comments: any text following a '%' character, up to an including the end of the line on which the '%'

occurs, is treated as a comment, of no semantic significance.

Variables in Parlog, as in all logic programming languages, are initially **unbound**. As computation proceeds, they may become **instantiated** by being bound to a term. Once instantiated, a variable cannot be bound to a different term. A term that contains no uninstantiated variables is said to be a **ground term**.

3.2 Operational Semantics

3.2.1 Evaluation Strategy

Parlog's evaluation strategy is based on a specialized form of resolution (Section 2.5.1) that features dataflow synchronization and guarded command non-determinacy (Section 2.2.3). The control strategy that it uses to control resolution features non-deterministic goal and clause selection. Resolution steps can be performed in any order or in parallel.

Parlog's mode declarations and commit operators constitute a control language (Section 2.5.1). They permit the programmer to influence the application of Parlog's control strategy. The mode declaration associated with a procedure identifies **input** and **output** arguments in clause heads. A '?' specifies that the head argument in the corresponding position is input; a '↑' specifies output.

Recall that a resolution step involves the attempted unification of the head of a clause and a goal. The result of this unification determines whether a resolution step succeeds or fails. Parlog specializes resolution in two respects. First, *input* clause head arguments are **matched** rather than unified with corresponding goal arguments. Matching attempts to unify two terms but **suspends** if it could only proceed by binding variables in the clause head (see Section 3.2.4). A Parlog resolution step may thus suspend as well as succeed or fail. This restricted form of unification provides the programmer with a means of controlling Parlog's otherwise non-deterministic control strategy. Reduction of goals can be delayed until data is available.

Once a Parlog clause is used to reduce a goal, there is no backtracking to try alternative clauses should subsequent evaluation leads to failure. The choice of clause to reduce a goal is thus a committed, potentially non-deterministic, choice. This feature of the language is termed **committed-choice non-determinism**.

Parlog's second specialization of resolution provides the programmer with additional control over which clause is used to reduce a goal. The evaluation of goals

in the guard of a clause (that is, those goals preceding any commit operator) is incorporated in the resolution step. Like matching, guard evaluation cannot bind variables in goal arguments and may suspend if goal arguments are not sufficiently instantiated. A Parlog clause cannot be used to reduce a goal until input matching *and* evaluation of the guard succeed.

Output mode clause head arguments are *unified* with corresponding goal arguments. However, this unification is performed after input matching and guard evaluation and after the committed choice to the clause.

3.2.2 A Computational Model

A number of computational models have been proposed for Parlog; see for example [Gregory, 1987; Foster and Taylor, 1988; Crammond, 1988]. The simplest views Parlog goals as **processes** and the state of a Parlog computation as a **process pool**. A query defines an initial process pool. Evaluation proceeds by repeatedly selecting a process from the pool and attempting to reduce it using the clauses in the associated procedure. A reduction attempt may succeed, fail or suspend.

A **process reduction** (that is, a Parlog resolution step) comprises two phases: *test* and *spawn*.

In the test phase, an attempt is made to find a clause capable of reducing the process. The **input** arguments of all clauses are **matched** with corresponding process arguments and guard processes are evaluated. Matching and guard evaluation may suspend if process arguments are not sufficiently instantiated. A clause is a **candidate** if both input matching and guard evaluation succeed. If no clause is a candidate and at least one clause has suspended, the reduction attempt **suspends** and the process is put back in the process pool. If no clause is a candidate and no clause suspends, the reduction attempt has **failed**. If one or more candidate clauses are found, the reduction attempt has **succeeded**. A candidate clause is non-deterministically selected and reduction proceeds to the spawn phase.

Spawn phase

In the spawn phase, the selected clause's **output** arguments are **unified** with corresponding process arguments. The clause's body goals are added to the process pool. Variables in both input and output process arguments may be bound, by output

unification, explicit calls to the Parlog unification primitive '=' and any other body call.

A Parlog computation terminates when:

- the process pool is empty, in which case the computation has **succeeded**.
- a reduction attempt or unification operation fails, in which case the computation has **failed**.
- there are no reducible processes, yet the process pool is non-empty, in which case the computation is **deadlocked**. (A process is **reducible** if a reduction attempt involving that process would not suspend. Deadlock thus indicates that all processes are suspended waiting for data).

This simple computational model is complicated by calls to user-defined procedures in guards and by other Parlog language features introduced in Section 3.2.4. These create a process hierarchy [Gregory, 1987]. However, as programs that use these language these features can be compiled to programs that do not (see Section 5.2.1), these complications are ignored here.

Program 3.1 will be used to illustrate Parlog's computational model. This defines a database with state subject to change over time. It may be executed with a query such as:

```
?- database([write(1, john), read(1, X), write(2, mary)]).
```

Its argument is a list of requests.

database(Rs) has the logical reading: Rs is a valid list of requests. Or, more precisely: Rs is a list of terms read(_,_) and write(_,_), in which each term read(K,V) is preceded in the sequence by a term write(K,V), with no write(K,V1) such that V≠V1 intervening. database([write(1,john),read(1,john)]) is a member of this relation; database([write(1,john), read(1,mary)]) is not. member(K,Db,V) has the logical reading: database Db contains the pair {K,V}.

The database can be viewed as a process which processes a list of requests Rs by maintaining a database Db of {Key,Value} pairs. When a message of the form read(Key,Value) is received, a member process is created to search the database for the appropriate key and retrieve the associated value (C2). (In discussions of this and subsequent programs, clause numbers (*Ci*) refer to the program text). Meanwhile, database recurses to process further requests. The member process inspects the database Db recursively (C5,6). If the key is found the associated value is returned (C5). If it is not present the member process fails. When a message write(Key, Value)

is received, the database simply recurses with an augmented database as its second argument (C3).

```
mode database(Rs?),  database(Rs?, Db?),  member(Key?, Db?, Value↑).

database(Rs) ← database(Rs, [ ]).          % Initialize empty database        (C1)

database([read(Key, Value) | Rs], Db) ←    % Receive read request:            (C2)
    member(Key, Db, Value),                %      look up Key in Db for Value;
    database(Rs, Db).                      %      recurse.
database([write(Key, Value) | Rs], Db) ←   % Receive write request:           (C3)
    database(Rs, [{Key, Value} | Db]).     %      recurse with new item.
database([ ], Db).                         % No more requests: terminate.     (C4)

member(Key, [{Key,Value}| Db], Value).     % Found.                           (C5)
member(Key, [{Key1,Data1}| Db], Value) ←   % Not found:                       (C6)
    Key =/= Key1 :                         %      check Key;
    member(Key, Db, Value).                %      continue.
```

Program 3.1 Simple database.

The database/2 procedure illustrates dataflow constraints. Its first argument has input mode: evaluation of a database process hence suspends until this argument is instantiated; that is, until a request is received.

The member procedure illustrates clause selection. Its first clause is selected if the Key specified in the message (the first argument) matches that of the first entry in the database. The second clause is selected if the guard test =/= succeeds, indicating that they do *not* match.

3.2.3 Standard Form

A procedure's mode declaration defines its translation to a standard form [Gregory, 1987]. In this form head arguments are all distinct variables and unification is performed by explicit calls to unification and matching primitives.

The standard form of a Parlog clause permits a simpler formulation of Parlog's operational semantics. A clause can be used to reduce a process if, when expressed in standard form, all its guard tests succeed. Once a clause is selected to reduce a goal, body goals are added to the process pool.

The standard form of the member procedure is:

```
member(A1, A2, A3) ←
    [(Key,Value)| Db] ⇐ A2, A1 == Key :
    A3 = Value.
member(A1, A2, Value) ←
    [(Key,Data)| Db] ⇐ A2, A1 =/= Key :
    member(Key, Db, Value).
```

Input arguments are compiled to calls to a matching primitive ⇐ in the guard. A call T1 ⇐ T2 attempts to unify T1 and T2 by binding variables in T1 so as to make T1 and T2 syntactically identical. It suspends if it could only proceed by binding variables in T2. Ultimately, it succeeds if and only if T1 and T2 unify; otherwise, it fails. *Multiple occurrences* of a variable in input mode positions are compiled to calls to an equality test == which suspends until it can determine whether or not its arguments are syntactically identical. It then succeeds if they are identical and fails if they are not. Two terms are syntactically identical if they are both the same variable, or the same constant, or are both tuples with the same arity and syntactically identical components. *Output arguments* are compiled to calls to a unification primitive = in the body of the clause. This performs general unification and may bind variables in either of its arguments.

The inequality primitive =/= is defined as follows: a call to =/= suspends until it can determine whether or not its arguments are syntactically identical. It then fails if they are equal and succeeds if they are not.

3.2.4 Other Language Features

Parlog's sequential clause search operator (';') may be used to constrain which clauses are tried when attempting to reduce a process. Clauses following such an operator in a procedure are only tried if all clauses preceding the operator have been tried and failed. The operator can be regarded as shorthand: a procedure which uses it has the same logical reading as a procedure in which the guards of clauses following the ';' are augmented with negated calls corresponding to the tests defined by input arguments and guards of all clauses preceding the ';'.

The sequential conjunction operator ('&') may be used to sequence process reduction. A sequential conjunction A & B delays evaluation of B until evaluation of A has successfully terminated. The primary application of the sequential conjunction

operator is to sequence calls to side-effecting primitives. As such primitives are primarily used by low-level systems programs (see Chapter 4), few application programs require this operator. Gregory [1987] proposes that sequential and parallel conjunction operators be used by a programmer or compiler to specify the granularity of parallelism in programs. This use of these operators is not considered herein. Instead, it is assumed that some system of program annotations is used for this purpose (see Section 6.3).

Parlog programs may also contain calls to a unary negation operator, not. A negated call not(P) has the logical reading 'P is false'. Like most logic programming languages, Parlog implements negation as 'failure to prove'. That is, a call not(P) succeeds if and only if evaluation of P fails. This is not necessarily equivalent to logical falsity, as Clark [1978] points out. However, it can be implemented efficiently and is sufficient for most purposes. Gregory [1987] discusses the semantics of Parlog's negation operator in detail.

Parlog's negation operator can be implemented in Parlog using sequential clause search, the Parlog primitive fail, which always fails, and a metalogical primitive call(P), which has the same logical reading as a call to its argument:

```
not(P) ← call(P) : fail;
not(P)
```

A call not(P) results in the evaluation of P in the guard of the first clause. If this succeeds, the clause commits and the call fails. If evaluation of P fails, on the other hand, the call not(P) is reduced using the second clause and hence succeeds.

3.2.5 Guard Safety

It was stated in Section 3.2.1 that guard evaluation cannot bind variables in process arguments. The reason for this restriction is that otherwise a guard could bind a process variable and subsequently fail; such a binding would be **incorrect**. Another potential problem is that multiple clauses for the same procedure could generate correct but **contradictory** bindings for the same process variable.

Parlog's operational semantics specifies that output unification is performed *after* guard evaluation has terminated. Guard evaluation cannot therefore bind variables in output arguments of a process. To ensure that variables in input arguments are not bound during guard evaluation, it is necessary to verify that guards are **safe**. A safe

guard does not bind variables in input arguments. It may only bind variables in output arguments or variables that occur in the guard but not the head of a clause.

Guard safety can be verified by a compile-time or run-time check. Efficiency considerations favor the former. While it does not appear possible to develop a compile-time check that will accept all programs with safe guards, algorithms that accept most such programs have been developed: see for example [Gregory, 1987]. These algorithms are relatively expensive and must be applied to an entire Parlog program. This hinders incremental compilation. Existing Parlog compilers therefore generally omit this check and rely instead on the programmer to write safe programs.

3.2.6 Justice Constraints

Parlog's operational semantics allows non-deterministic process and clause selection. This under-specification of process and clause selection strategies facilitates implementation, as any convenient strategies may be used. However, it makes it difficult to reason about program behaviour. For example, consider the program:

```
mode q(?), p(↑).
q(halt).                    p(halt).
q(X) ← q(X).
```

Intuitively, it might be expected that evaluation of the query

```
?- p(X), q(X).
```

would eventually terminate. Reduction of p(X) should bind X to halt; the process q(X) can then be reduced with the first clause of the procedure q/1. However, Parlog's operational semantics permits an implementation to never reduce p(X), or to repeatedly reduce the process q(X) with the second clause for the procedure q/1. Evaluation may therefore never terminate.

If termination of this query is to be guaranteed, the Parlog implementation that executes it must possess two properties: **And-justice** and **Or-justice**. View a Parlog clause as defining a transition from a process to a set of new processes. If a clause can be used to reduce a process, the transition defined by that clause is said to be *enabled*. Then these properties may be defined as follows:

And-justice: a transition that is continuously enabled will eventually be taken.
Or-justice: a transition that is infinitely often enabled will eventually be taken.

And-justice requires that if a process is continuously reducible, then it will eventually be reduced. Or-justice is stronger. It requires that if the same process is reduced repeatedly, then each clause in the associated procedure that is capable of reducing the process will eventually be used to reduce it. In the example, And-justice guarantees that the process p(X) will eventually be reduced. Or-justice guarantees that once p(X) is reduced, binding X to the constant halt, the first clause of the relation q is eventually selected to reduce the process q(halt), thus terminating evaluation of the query.

A Parlog implementation may be And-just but not Or-just. To guarantee termination, the above example may then be rewritten as:

```
mode q(?), p(↑).
q(halt).                p(halt).
q(X) ← var(X) : q(X).
```

A call to the Parlog primitive var(X) is generally defined to have the semantics: "X is a variable at the time of call". In the example, the call var(X) ensures that the second clause is not used to reduce the process q(X) once X is instantiated to the constant halt.

On a multiprocessor, message propagation delays mean that it may not be possible to guarantee that a variable X is unbound at the time a call var(X) succeeds. This observation has prompted the proposal of an alternative semantics for var. This requires that a call var(X) succeed only a finite number of times after a variable X is instantiated. This implies that bindings to variables that have been tested using the var primitive are propagated to other nodes in a multiprocessor *within a finite time*. An implementation of the var primitive that has this property may be termed *Var-just*.

And-, Or- and Var-justice are useful properties for an implementation to possess. They permit certain qualitative questions about program behaviour (such as: *"will the query* q(X), p(X) *eventually terminate?"*) to be answered. However, they complicate Parlog's otherwise simple operational model. This can lead to difficulties when implementing Parlog on parallel architectures.

Existing implementations of Parlog and other parallel logic languages generally provide And-justice and Var-justice but not Or-justice.

3.3 Programming in Parlog

This section describes a number of important Parlog programming techniques. Many of these derive from the particular characteristics of Parlog's logical variable, which is seen to be a powerful feature of the language. Program 3.2 is used to illustrate the discussion. This is a simplified version of a real train control program [Foster, 1988]. It controls a single train that must move between two stations on a single track in response to requests for transportation (Figure 3.1). Note that the program is constructed so as to illustrate Parlog programming techniques, rather than to present an optimal solution to this problem.

Figure 3.1 Simple train system.

Upon receiving a request, the train control program must determine the sequence of commands required to service the request and transmit those commands to a track controller device in the correct sequence and at the correct time. (For brevity, the part of the program that transmits the commands to the controller — the device procedure — is not given). The controller recognizes commands on(w) and on(e), which enable power in the track in a westerly and easterly direction, respectively, and off, which disables power. It also generates interrupts when a train passes west and east sensors (denoted ▼ in Figure 3.1). For example, to service a request for transportation from east to west, when the train is initially at the east station, it is necessary to generate the command on(w), wait for the west sensor to be enabled, and then generate the command off. If the train is initially at the west station, it is necessary to generate commands to move the train to the east station to collect a passenger and then to return to the west station.

3.3.1 Process Networks

A conjunction of Parlog goals may be interpreted as defining a process network. Processes execute concurrently and communicate by binding shared variables, which can be regarded as defining communication channels. As Parlog variables can only be bound to a single value, there is generally a single producer for each shared variable and one or more consumers.

```
mode train(Location?, Requests?), control(Control?, Requests?), device(Device?),
     train(Location?, Device↑, Control↑), train(Location?, Destination?, Device↑, Control↑),
     actions(From?, To?, Actions↑), perform(Actions?, Device↑, Device1?, Next↑),
     perform(Actions?, Device↑, Device1?, Next↑, Synch?).

train(L, Rs) ← train(L, Ds, Cs), control(Cs, Rs), device(Ds).                    (C1)

control( [ To |Cs], [T |Rs]) ← To = T, control(Cs, Rs).                          (C2)
control( [ To |Cs], [ ]) ← To = halt.                                            (C3)

train(L, Ds, [To |Cs]) ← train(L, To, Ds, Cs).                                   (C4)

train(From, halt, [ ], [ ]).                                                     (C5)
train(From, To, Ds, Cs) ←                                                        (C6)
     To =/= halt :
     actions(From, To, As), perform(As, Ds, Ds1, L), train(L, Ds1, Cs).

actions(w, e, [on(e), wait(e), off, done(e)]).                                   (C7)
actions(w, w, [on(e), wait(e), off, on(w), wait(w), off, done(w)]).              (C8)
actions(e, w, [on(w), wait(w), off, done(w)]).                                   (C9)
actions(e,e, [on(w), wait(w), off, on(e), wait(e), off, done(e)]).               (C10)

perform([on(D) |As], [on(D) |Ds], Ds1, L) ← perform(As, Ds, Ds1, L).             (C11)
perform([off |As], [off |Ds], Ds1, L) ← perform(As, Ds, Ds1, L).                 (C12)
perform([wait(D) |As], [wait(D, S) |Ds], Ds1, L) ← perform(As, Ds, Ds1, L, S).   (C13)
perform([done(L) |As], Ds, Ds, L).                                               (C14)

perform(As, Ds, Ds1, L, proceed) ← perform(As, Ds, Ds1, L).                      (C15)
```

Program 3.2 Train controller.

As processes reduce to new processes, the process network is reconfigured. It is useful to think of a process which reduces repeatedly to a recursive call to the same procedure as a single process that persists in time: a long-lived or **perpetual process**. For example, an initial call to Program 3.2:

```
?- train(west, [east, west, west])
```

(the arguments indicate the train's initial position and a list of requests for transportation, respectively) generates a network of three perpetual processes: train, control and device (C1). The train process is actually defined by two mutually

recursive procedures, train/3 (C4) and train/4 (C5,6), but may still be thought of as a single process. Shared variables Ds and Cs provide for communication between these processes. Modes determine the direction of communication: train generates bindings for both variables (mode ↑) which control and device consume (mode ?).

The network of perpetual processes can be represented graphically, as shown in Figure 3.2. The circles in this figure represent processes and the labelled lines shared variables. The arrows on the lines indicate in which direction the variables are used to pass information.

Figure 3.2 Train process network.

3.3.2 Stream Communication

A process may instantiate a variable to a structured term containing a new variable. Later, it may instantiate the new variable to a similar structure, containing a further variable. Especially if list structures are used, this incremental construction of terms is termed *stream communication*: a single shared variable is used to pass a **stream** of values between two processes.

Clause C4 provides an example of stream communication. This clause reduces the train process to a call to train/4 and in so doing generates a stream communication to control. The stream communication consists of a single variable To, a request for a destination for the train's next journey. The new train process takes To and the new stream variable Cs as arguments.

3.3.3 Back Communication

In Parlog, a single shared variable can be used to implement two way communication. A producer process may generate a structure containing unbound variables. If a consumer process subsequently instantiates these variables, this causes a 'back communication', from the consumer to the producer. This may affect the subsequent behaviour of the producer. (This elegant use of the logical variable was first described by Shapiro [1983], who termed it 'incomplete messages').

For example, the control process consumes a stream of variables generated by train. Each variable received is unified using Parlog's unification primitive, '=', with the next element on a list of destinations (C2) or, if this list is empty, with the constant halt (C3). This unification has the effect of communicating a new destination back to the train process, which cannot reduce until the variable To is bound (C5,6).

3.3.4 Process Termination

Reducing a process using a clause with no body goals has the effect of terminating that process. For example, the control process terminates when train asks for a destination and all requests have been processed (C3). As it terminates, control communicates the destination halt to train. This causes train to terminate and to close the device stream (C5).

3.3.5 Process Creation

Reducing a process using a clause with more than one goal in the body creates new processes. A spectrum of more or less long-lived or transitory processes may be created. For example, once train receives a destination from control, it creates two new processes, actions and perform (C6). actions is a short-lived process which determines the sequence of actions that must be performed to make a journey from some location to a destination and immediately terminates (C7-10). perform is longer-lived: it generates messages to device for each action (C11-13), before terminating (C14).

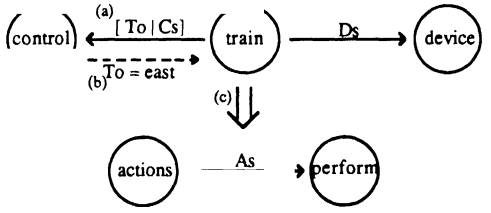


Figure 3.3 Communication and process creation.

Figure 3.3 illustrates stream communication, back communication and the creation of short-lived processes. A stream communication from train to control (a) is followed

by a back communication from control to train (b). This leads to the creation of processes actions and perform (c).

3.3.6 State and State Change

A process which calls itself recursively with different arguments can be regarded as possessing an internal state, subject to change over time. In the example, both the train and control processes have an internal state: a location and a list of destinations, respectively. As computation progresses, this state changes: train recurses with a different location and control recurses with a reduced set of destinations.

3.3.7 Encapsulation of Devices

Streams and processes are a convenient mechanism for interfacing to the physical world. Both input and output devices may be encapsulated inside processes which generate and consume streams of messages, respectively. These processes may use side-effecting primitives to perform the input or output operations, but this is hidden from the rest of the Parlog program.

For example, the device process (program not presented) encapsulates the track controller. It consumes a stream of requests on(w), on(e) and off, which it translates into commands to the track controller, and wait(w,S) and wait(e,S), which it processes by binding the variable S when the west or east sensor next generates an interrupt, respectively. An implementation of device might process these requests using command and interrupt primitives, which set appropriate device registers and suspend until an interrupt occurs, respectively.

3.3.8 Synchronization

Previous sections have shown how shared logical variables can be used to communicate values between processes. Another important, related use of shared logical variables is process synchronization. Parlog reduction is eager. The reduction of processes that side-effect their environment must be coordinated if side-effects are to be correctly sequenced. This can be achieved by using bindings to shared logical variables to synchronize process reductions. Variables used for this purpose may be termed **synchronization variables.**

A train must not send an off signal to the track controller until it has passed a sensor

indicating that it has arrived at a station. This synchronization requirement is indicated in the 'plan' (C7-10) by keywords wait(w) and wait(e). The perform process handles a wait(D) keyword by sending a message wait(D,S) message to the track controller (C13) and recursing to call perform/5 (C15). perform/5 cannot reduce until its fifth argument, the variable S, is bound to the constant proceed by the device controller process, indicating that the train has passed the appropriate sensor. S is a synchronization variable used to delay the reduction of perform/5 (and hence the generation of an off signal) until the sensor has been triggered.

3.3.9 Non-determinism

When two or more clauses can be used to reduce a process, Parlog's operational semantics do not specify which is to be used. An implementation may select the first clause for which sufficient data is available to permit reduction to proceed. This permits the programming of systems that process information as it becomes available in real time.

```
mode merge(Xs?, Ys?, Zs↑).

merge([X |Xs], Ys, [X |Zs]) ← merge(Xs, Ys, Zs).
merge(Xs, [Y |Ys], [Y |Zs]) ← merge(Xs, Ys, Zs).
merge([ ], Ys, Ys).
merge(Xs, [ ], Xs).
```

Program 3.3 Merge.

A good example of a non-deterministic Parlog program is the merge relation (Program 3.3). merge(Xs, Ys, Zs) has the logical reading: if Xs and Ys are lists, Zs is some interleaving of Xs and Ys.

If, when a merge process is reduced, only its second argument is bound to a list, then the second clause of the merge procedure is used to reduce that process. This selects the item available on the second argument. If only the first argument is so bound the first clause is used. A merge process thus tends to select items from its two input streams as they become available, if it is selected for reduction at least as frequently as items are produced. Figure 3.4 represents a merge process.

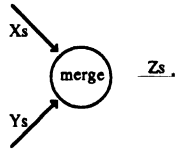


Figure 3.4 Merge.

Program 3.2 is totally deterministic. However, a more realistic program that controls a track layout with several trains, controllers and sources of requests would have non-deterministic components. For example, a program that controls two trains might be defined as follows:

```
train(L1,L2,Rs) ←
        control(Cs,Rs), merge(Cs1,Cs2,Cs), merge(Ds1, Ds2, Ds), device(Ds),
        train(L1,Cs1,Ds1), train(L2,Cs2,Ds2).
```

Figure 3.5 represents the process network that results. Each train process generates, deterministically but asynchronously, streams of device commands and requests for destinations. The merge processes pass on some intermingling of the two command and request streams to the device and control process respectively.

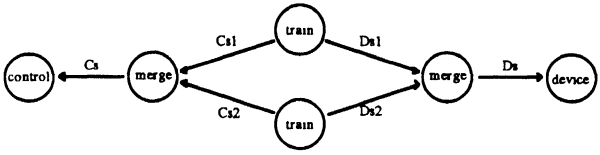


Figure 3.5 Non-deterministic train network.

3.3.10 Multiprocessors

The implementation of Parlog on multiprocessors is quite simple (see Chapter 5). Processes are located on the various nodes. The language implementation extends unification to deal with variables located on other nodes. Testing or binding a variable shared by processes located on different nodes has the effect of communicating a value between these processes. Unification thus becomes a communication as well as a data construction and pattern matching mechanism.

For example, if the program illustrated in Figure 3.5 were to be executed on a multiprocessor, each perpetual process could be located on a different node. The

variables Cs1, Cs2, Cs, Ds1, Ds2 and Ds then represent *physical* communication channels.

3.3.11 Logical Reading

Most of the procedures in Program 3.2 have a straightforward logical reading as statements about relationships in the model train system. For example:

actions(F,T,As): the actions As must be performed to move from location F to T.

perform(As,Ds,Ds1,L): performing actions As leaves a train at location L and generates as device commands the difference between the lists Ds and Ds1.

train(L,Ds,Cs): Ds is the device commands required to serve requests Cs when originally at location L.

3.4 The Control Metacall

Clark and Gregory [1984] extend Parlog with a metalogical primitive termed a **control metacall**. This permits Parlog programs to initiate, monitor and control the execution of other Parlog programs. A call to this primitive has the general form:

call(Goal, Status, Control)

and denotes the controlled execution of Goal, which is assumed to be a term representing a relation call. A call to the control metacall primitive always succeeds, unless the control argument is bound to an incorrect value. The result of evaluating Goal is reported by unification with the Status argument. Possible results include the constants succeeded, failed and stopped. A concurrently executing process may bind the Control argument to the constant stop. This has the effect of terminating the evaluation of Goal and, once evaluation has terminated, unifying Status with stopped. The Control argument may also be bound to a list with the constants suspend or continue as its head; this suspends or resumes the evaluation of Goal, respectively, and causes the Status argument to be unified with an equivalent list structure upon completion of this action.

Program 3.4 illustrates the use of this primitive. This implements a simple command interpreter or 'shell' that processes a stream of requests bg(G,R), fg(G,R) and

abort by initiating execution of a goal G in the foreground (C3) or background (C4), or aborting execution of the current foreground command (C2), respectively. In the first two cases the variable R is bound to the result of this execution. Background goals execute independently. A foreground goal delays initiation of further goals until it has terminated.

```
mode shell(Goals?), shell(Goals?, Status?, Control↑).

shell(Gs) ← shell(Gs, go, C).                                              (C1)

shell([abort |Gs], S, stop) ← shell(Gs, S, C).                             (C2)
shell([ fg(Goal, S1) |Gs], S, C) ← data(S) : call(Goal, S1, C1), shell(Gs, S1, C1).   (C3)
shell([bg(Goal, S1) |Gs], S, C) ← data(S) : call(Goal, S1, C1), shell(Gs, go, C).     (C4)
shell([ ], S, C).        (C5)
```

Program 3.4 Simple shell.

shell/3 is called recursively to process a list of requests. It processes a fg or bg request by initiating execution of the specified goal using the control metacall, *if* its second argument (assumed to be the current foreground goal's status variable) is non-variable (C3,4). This indicates that the current foreground process, if any, has terminated. The initial call to shell/3 has its second argument bound to the constant go (C1), as do recursive calls following initiation of background goals (C4). This permits reduction to proceed immediately in these cases. The test for a non-variable term is performed using Parlog's data primitive (C3,4), which suspends until its argument is bound and then succeeds.

The second component of fg and bg requests, the variable R, is passed to the metacall used to initiate the associated goals as the metacall's status variable (C3,4). It is hence unified with succeeded, failed or stopped when evaluation of the goal terminates.

shell/3 processes abort requests by unifying its third argument (the control stream of the current foreground goal, if any) with the constant stop (C2). This aborts execution of any current foreground goal.

Figure 3.6 illustrates the execution of this shell. Note that in this and subsequent figures, metacalls are represented as boxes to distinguish them from processes, which are pictured as circles. In (a), the shell is initiated with three requests on its input

stream. In (b), the first two requests are processed and a background and a foreground goal are initiated. In (c), G2 terminates, permitting the second foreground goal, G3, to start executing. At about the same time, the shell receives an abort request; G3 is thus aborted, leaving only G1 executing in (d).

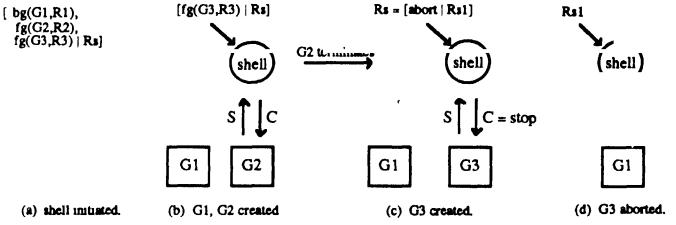


Figure 3.6 Execution of simple shell.

shell(Gs) has the *metalogical* reading (see Section 3.5.5): Gs is a list of terms fg(G,R), bg(G,R) and abort, where in each case R may be the result (succeeded or failed) of evaluating G or, in the case of a fg term followed immediately by an abort term, stopped.

The control metacall is central to the use of Parlog for systems programming, as it provides a succinct and elegant representation of the important notion of **task**. In conventional systems, a task may defined to be a *thread of control* or *process* created to perform some activity, such as edit a file, read mail or service device interrupts. In Parlog, the corresponding entity is the pool of processes created during execution of a goal. On a multiprocessor, these may be distributed over many processors; nevertheless, it is necessary to be able to detect changes in their status and control their execution as if they were, in some respects, a single entity. The metacall provides this capability.

3.5 Parlog as a Declarative Language

3.5.1 Correctness and Completeness

Parlog programs (such as Programs 3.1, 3.2 and 3.3) can be read as logical axioms describing relations between entities in some problem domain (see Sections 3.2.2, 3.3.11).

Parlog's evaluation strategy, being based on resolution, is **correct**: any solution computed by a Parlog computation from a Parlog program that does not use metalogical

operators such as var or the control metacall is a solution according to the program's declarative semantics.

Because of its committed-choice non-determinism, Parlog is **incomplete**. Each time Parlog evaluation commits to a single candidate clause, it irrevocably abandons other potential solutions. As a result, Parlog evaluation may sometimes result in failure even when solutions exist according to declarative semantics. The set of solutions that Parlog can compute is in general a subset of the set of solutions implied by the declarative reading of a program.

The declarative reading of a Parlog program thus provides a useful, though informal, check of its correctness. It is good Parlog programming style to enhance this utility by writing programs in which the two sets of solutions are equivalent. One component of good style is what Gregory [1987] calls the **sufficient guards law**. This states that Parlog procedures should be written so that guards to each clause guarantee that if that clause is selected to reduce a goal, either:

- a solution to the call can be computed using this clause, or
- no solution can be found using any other clause

This ensures that a Parlog computation returns a solution to a query, if a solution exists.

3.5.2 Program Analysis and Transformation

Program analysis and transformation techniques developed in the broader context of declarative language research have been successfully applied to parallel logic languages. For example, partial evaluation [Komorowski, 1981; Safra and Shapiro, 1986; Huntbach, 1987], fold/unfold transformations [Burstall and Darlington, 1977] and enhanced metainterpreters [Kowalski, 1979; Safra and Shapiro, 1986]. Incompleteness and dataflow constraints cause difficulties, but these can generally be overcome.

Program transformation seeks to modify some property of a program (such as efficiency) whilst maintaining other properties (such as the input/output relation that it computes) invariant. A metainterpreter for a language is an interpreter for the language written in the language. Metainterpreters have been shown to be powerful programming tools, facilitating debugging and permitting experimentation with

alternative evaluation strategies. Partial evaluation techniques can be applied to enhanced metainterpreters to compile additional functionality into programs. Several Parlog metainterpreters are presented in Appendix I.

3.5.3 Consequences of Incompleteness

Parlog's committed-choice non-determinism means that the Parlog programmer cannot be vague about how solutions are to be computed; he must specify explicitly the algorithm to be used. This is in essence what the sufficient guards property requires of him. In contrast, Prolog permits the programmer to describe conditions that a solution must satisfy; Prolog then explores alternatives by a backtracking search. This don't-know non-determinism accounts for much of the elegance and succinctness of Prolog programs constructed to solve problems in (for example) natural language parsing.

Several techniques for integrating don't-know non-determinism and the search that it implies into parallel logic languages have been proposed. Clark and Gregory [1986] propose a set constructor primitive for Parlog that performs Prolog-like evaluation. Ueda [1986] proposes compiling programs intended to explore all solutions to a query into parallel logic programs that perform the search explicitly. Somogyi [1987] proposes a system of declarations that permit a compiler to verify that a given procedure is **stable**: that is, that it provides exactly one binding for any output variable. He argues that this can provide the determinacy required for systems programming, while preserving the search ability of Prolog internally.

It can be argued that Parlog's lack of don't-know non-determinism is not of great importance when the language is used for systems programming. This is because the systems programmer generally has a definite algorithm in mind: he knows not only *what* solution he wants computed, but *how* it should be computed.

3.5.4 Restricted Modes of Use

The modes associated with Parlog relations restrict the argument patterns that they can be called with. In effect, Parlog's specialized unification means that unification may suspend forever instead of succeeding or failing. For example, consider Program 3.1, and the query:

```
?- member(X, [{1, john}], john)
```

This should succeed, binding X to 1, as member(1,[{1,john}],john) is a member of the relation defined by this program. However, Parlog evaluation results in indefinite suspension rather than success.

To understand the pragmatic implications of this restriction, it is important to note that logic programs can often be divided into distinct *database* and *algorithmic* components. The database component consists of assertions (clauses with no body goals) and simple rules that describe a problem domain. The algorithmic component consists of more complex rules that describe how this data is to be used. For example, in Program 3.2 the procedure actions/3 may be classified as database and the other procedures as algorithmic.

Algorithmic procedures are not generally intended to be used in more than one mode of use. In any event, existing logic languages rarely permit efficient use in more than one mode. Database procedures, on the other hand, are used very effectively in different modes in languages such as Prolog. This permits a database to be specified independently of its intended use. This encourages generality and facilitates reuse.

It is possible to write Parlog programs that can be used in more than one mode. Additional clauses which explicitly perform the required input matching and unification operations must be added to a program. However, this approach produces large, clumsy programs. An alternative approach is taken in P-Prolog [Yang and Aiso, 1986]. This language uses a more general synchronization mechanism than Parlog. Reduction of a P-Prolog **exclusive relation** is delayed until only a single clause is capable of reducing a call. This permits programs to be used in alternative modes. It is not however clear that P-Prolog can be implemented efficiently.

Systems programs generally have a significant algorithmic component. Parlog's mode restriction may thus not be a serious problem in this type of application. Also, the simplicity of most database procedures means that when they must be used in more than one mode, it is not an onerous task to duplicate a procedure.

3.5.5 The Control Metacall

The control metacall primitive (Section 3.4), denoting as it does the monitoring and control of computation, lacks any *logical* reading. However, it can be given a *metalogical* reading in that it permits metalevel programs to talk about the execution of other, object level programs. The shell presented on Program 3.4 is an example of a program with a metalogical reading.

The metacall can be given a logical reading in another sense. It is possible to provide a specification, in Parlog, for its functionality. This specification takes the form of a metainterpreter for Parlog, augmented with code that detects termination and permits control. Appendix I presents such a specification. A call to the metacall is logically equivalent to a call to this specification.

A potential problem with the control metacall's semantics should be noted. The theory of metaprogramming requires that metaprograms manipulate representations of object programs, rather than the programs themselves [Bowen and Kowalski, 1982]. Parlog, unlike other systems that explicitly support metaprogramming (such as metaProlog [Bowen, 1985]) does not provide separate representations for meta and object level terms. A metaprogram that examines an object level variable thus sees a variable, rather than a representation of a variable. It is thus possible, though in general logically meaningless, for the metaprogram to instantiate the variable. This can lead to anomalous behaviour. For example, as Ueda [1986] points out, the Parlog conjunction:

```
call(X=1, S, C), X=2
```

gives different results depending on the order in which the goals are evaluated. The computation either fails, if X is bound to 1 by the first goal, or succeeds, binding S to failed, if X is bound to 2 by the second goal.

The problem is that the same variable occurs at both the object and metalevels; the conjunction thus has no logical or metalogical meaning.

One solution to this problem is to ensure that meta and object level programs do not share variables. This restriction can be enforced in Parlog using simple compile- and run-time checks: for example, ensuring that all terms passed to programs invoked using the control metacall are either ground at the time of call or are not shared with metaprograms. However, these restrictions forbid many useful applications of shared variables, which can in any event frequently be given a metalogical reading. Methodologies for avoiding conflicts when meta and object level programs share variables are discussed in Chapter 4.

3.5.6 Limitations

One way of understanding a computer program is to execute it on a computer and observe its behaviour. Programming language operational semantics define the

meaning of a program to be some abstraction of its behaviour when executed. The abstractions they employ are useful if the details that they hide can easily be ignored.

The declarative semantics of logic programs effectively defines the meaning of a program to be the input-output relation that it computes. This **transformational** view of computation is very abstract and hence particularly easy to analyse and understand. However, declarative semantics has its limitations as a semantics for Parlog, because it conceals timing information that may be necessary for an understanding of a Parlog program's behaviour.

Parlog programs are frequently **reactive** in nature. Reactive programs may be defined as programs that maintain an ongoing interaction with their environment [Pnueli, 1986]. Parlog programs interact with their environment by binding variables. The meaning of a reactive Parlog program is characterized not only by the variable bindings it produces but also by the sequence and timing of these bindings.

Program 3.2 can be used to illustrate the consequences of the reactive nature of Parlog programs. As noted in Section 3.3.11, relations in this program have a simple logical reading about relationships in the model train world: train/3, for example, defines the commands required to process a list of requests. However, the logical reading says nothing about the time at which commands are generated. Thus, though the logical reading provides a check on the correctness of an important aspect of the program, it is not sufficient to verify that the program behaves correctly. Other techniques are required to specify and verify properties such as the temporal ordering of communications, termination, lack of starvation, etc. Temporal logic is a possible specification tool [Pnueli, 1986]. A formal semantics for parallel logic languages is required before verification techniques can be developed.

3.6 Parlog and Systems Programming

Recall the requirements for a high-level systems programming language listed in Section 2.1. Parlog's support for concurrency, communication, synchronization and non-determinism has been illustrated in this chapter. Its computational model uses the same two simple, uniform mechanisms — process reduction and unification — whether executing on a single processor or a multiprocessor. Other requirements, such as the ability to encapsulate hardware, language support for protection, language extensibility and ease of implementation are investigated in the chapters that follow.

CHAPTER 4.

Operating System Design

A typical architecture for a multiprocessor language-based operating system locates a small hardware or machine language **kernel** at each node. This kernel provides run-time support for the systems programming language. It also implements basic device control, interrupt handling and process scheduling mechanisms. These are made available to OS programs in terms of high-level language constructs.

Mechanisms implemented in the kernel are used to construct basic **operating system** services. These may also be replicated at each node or, in the case of higher-level services, located only on some subset of all nodes. Application or **user-level** programs are constructed using the services provided by the operating system. Figure 4.1 illustrates this architecture and relates it to the structure of this monograph.

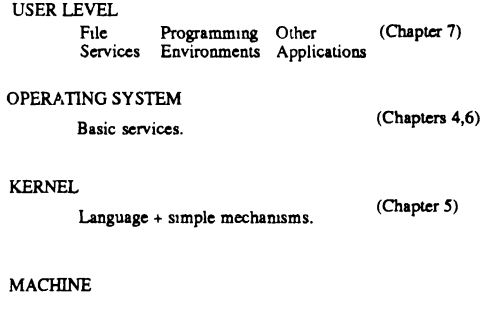


Figure 4.1 Operating system architecture.

This chapter deals with the *operating system* level in this architecture. The first section of the chapter identifies important issues in OS design. Subsequent sections consider how these issues can be handled in Parlog.

Section 4.2 considers how kernel mechanisms are accessed by OS programs. Minor extensions to the Parlog language that facilitate access to kernel mechanisms are motivated and described. The sections that follow deal with the provision of services, communication with the OS, system robustness, naming and protection. Section 4.7 describes a simple Parlog OS that integrates components introduced in preceding

sections. The final section looks briefly at how user-level programs can construct new abstractions using Parlog OS services.

4.1 Issues in Operating System Design

Kernel Functionality

The kernel of a language-based OS must either implement basic OS functions such as device control, exception handling and resource management directly or provide simpler mechanisms that permit these functions to be implemented in the high-level language.

Kernel design must resolve four conflicting requirements: functionality, efficiency, flexibility and simplicity. A kernel must provide support for all functions required by an operating system. Functions are more efficient when implemented in the kernel. However, as the kernel is difficult to modify, they are generally less flexible. A simple (and hence compact) kernel increases the proportion of high-level language code and minimizes the amount of code that must be duplicated on each node. It is also likely to be more reliable.

Wulf *et al.* [1981] propose that the performance/flexibility conflict be resolved by implementing **mechanisms** in the kernel while providing higher-level programs with the ability to define the **policies** that determine how these mechanisms are used. For example, a kernel may implement memory management functions but allow OS programs to specify paging strategies. Of course, absolute policy/mechanism separation is not achievable: the mechanisms implemented determine the policies that can be supported. Also, generality may conflict with the need for compactness. The Hydra kernel, for example, provides a high degree of policy/mechanism separation. However, it consists of about 130,000 words of object code [Wulf *et al.*, 1981].

The high-level systems programming language must be provided with an interface to the mechanisms implemented in the kernel. This may be achieved by adding new primitive operations to the language. For example, CCN-Pascal [Joseph *et al.*, 1984], though a high-level language, supports a primitive varaddr which it uses to access hardware locations directly. Alternatively, certain language operations may be interpreted in special ways in certain contexts. For example, in a language based on message passing, messages sent to certain reserved destinations can be interpreted as instructions to hardware. Similarly, external events may be translated into messages to OS components.

The interfacing of Parlog OS programs to kernel mechanisms is considered in Section 4.2. The implementation of a kernel that provides these mechanisms is discussed in Chapter 5.

Operating System Services

Services may either be made an integral part of the OS or may be supported as application programs. These two approaches have different performance/flexibility tradeoffs. Some of these are discussed in [Tanenbaum and van Renesse, 1985]. For example, a file service may be made an integral part of a OS. This can be expected to be efficient. However, it is then difficult for users to define alternative file services. Alternatively, the OS may only provide a simple block I/O service. Application programs may then build on this to provide a Unix style file service, a robust file service, and so forth.

The design and implementation of Parlog services is discussed in Section 4.3.

Communication with the Operating System

An efficient and safe mechanism is required to permit application programs to request OS services. On uniprocessors, such **system calls** are generally implemented using a **supervisor call** instruction, which causes a context switch to a privileged supervisor program. The supervisor processes the application program request, then returns control to the user program.

The supervisor call approach is inappropriate in multiprocessors, as the service requested may be located on another node. It is unacceptable to block the requesting program while the request is serviced. An alternative approach is to provide application programs with the ability to generate messages to system services. They can then continue processing until a reply is received.

Parlog application program access to Parlog OS services is discussed in Section 4.4.

Robustness

No user program action should be able to cause failure of OS components. To ensure this, OS programs may be designed to be **robust**; that is, to cope with unexpected requests. Robustness generally conflicts with performance, so an alternative approach is to ensure that only trusted agents can access OS services.

Section 4.5 describes an approach to the design and implementation of robust Parlog OS services.

Naming and Protection

Names are the means by which application programs access OS resources. Names are generally symbolic: that is, they are mapped by the OS onto some other domain (such as the domain of hardware addresses) before the resource is accessed. Naming is related to protection, as restrictions on both possession of names and the mappings applied to them can be used to control access to resources.

Names can be resolved at run-time, compile-time or at intermediate stages such as load-time. Run-time resolution of names (referred to as late binding) is more flexible but also more expensive than compile-time resolution (early binding).

The specification of naming schemes in Parlog is considered in Section 4.6.

Resource Management

Resource management embraces the problems of memory management, processor scheduling and deadlock detection and avoidance [Peterson and Silberschatz, 1983].

Extensions to Parlog that permit Parlog OS programs to specify processor scheduling and memory management policies are described in Section 4.2. Kernel support for processor scheduling is described in Chapter 5.

A system is reliable, or **fault tolerant**, if it is able to deliver a minimum level of service in the face of hardware and software failure. Reliability may be achieved by **redundancy** and by implementing **atomic actions**. Redundancy duplicates critical services and data. Atomic actions ensure that hardware failure does not lead to inconsistent states.

A Parlog implementation of atomic actions is described in Chapter 7. However, in general the problem of reliability is not considered herein.

Multiprocessors

None of the issues noted above are peculiar to multiprocessors. Yet most become more complex on parallel computers. For example, resource management algorithms must consider not only uniprocessor (*local*) scheduling but also the allocation of processors to tasks (*global* scheduling) [Wang and Morris, 1985]. It is not in general possible to have a global view of resource availability on multiprocessors [Tanenbaum and van Renesse, 1985].

Problems of multiprocessor OS design are considered in Chapter 6.

4.2 Kernel Functionality

This section describes a number of mechanisms that need to be supported by a Parlog OS kernel and discusses how these mechanisms can be accessed by Parlog programs. Minor extensions to Parlog that provide this access are defined. The design and implementation of the kernel itself is considered in Chapter 5.

4.2.1 Types of Interface

A Parlog OS kernel must support interfaces that (a) permit Parlog OS programs to invoke kernel mechanisms and (b) notify Parlog OS programs of significant events. Both types of interface can be based on either of Parlog's two basic operations: *unification* and *procedure call.*

(a) Invocation of Kernel Mechanisms

Unification-based interfaces to kernel mechanisms provide special variables which, when instantiated, invoke these mechanisms. For example, a terminal driver may be invoked by incrementally instantiating a special variable to a stream of terms representing terminal control codes.

Procedure call-based interfaces to kernel mechanisms provide primitive procedures which, when called, invoke these mechanisms. For example, a primitive which sets device registers in order to control a terminal. Procedure call-based interfaces are somewhat less elegant than unification-based interfaces, as they involve side-effecting primitives. However, they permit more functionality to be implemented in Parlog, and are hence preferred. As calls to primitives can be encapsulated inside services — Parlog processes that respond to streams of requests (see Section 4.3) — a unification-based interface can be simulated using a procedure call-based interface.

(b) Notification of Events

Unification-based interfaces to events notify Parlog OS processes of events by instantiating special variables. For example, keyboard input and clock ticks may be made available as streams of terms, constructed incrementally as data becomes available. This type of interface is elegant and easy to implement. It is used in the Parlog OS described in Section 4.7. A limitation of this type of interface is that it does not guarantee a timely response to events. The occurrence of an event makes any processes suspended waiting for it reducible. However, it does not guarantee when

they will be reduced. This depends on the scheduling strategy used by the Parlog implementation.

Procedure call-based interfaces require that the kernel execute certain Parlog procedures when events occur. These can be regarded as 'interrupt service routines'. This approach can ensure a rapid response to events, as procedures are invoked immediately an event occurs. However, as these procedures are invoked as independent goals, they cannot share variables with OS processes. This prevents them from communicating with the rest of the OS. This type of interface is not therefore very useful.

In Section 5.4, a hybrid interface to events is described. The kernel provides a unification-based interface to certain events (for example, timer interrupts). It also provides a specialized process scheduling strategy that switches to executing the Parlog process that consumes this event stream when certain events occur. This ensures that events are processed as soon as possible.

Parlog's control metacall permits a third type of interface to the kernel. The metacall's status stream can be used to signal events that occur while executing a task. The metacall's control stream can be used to invoke kernel mechanisms with respect to the task. Examples of these types of interface follow.

4.2.2 Exceptions

An **exception** is an unusual occurrence that requires special attention. For example, calls to undefined relations, floating point overflow and memory parity errors. Typically, exceptions are detected by hardware and are translated into interrupts that are trapped by the OS kernel. The kernel must generally signal exceptions in the context (for example, the task) in which they occur.

It is inconvenient to place all exception handling in the OS kernel. An interface to Parlog is thus required that permits OS programs to be notified of exceptions. A simple extension to Parlog's *control metacall*, originally proposed in [Foster, 1987a], provides this facility. Recall that a call to the control metacall initiates execution of another program as a separate task. Status and control streams then permit this task to be monitored and controlled (Section 3.4).

An exception that occurs during the execution of a task leads to the generation of an **exception message** on the status stream of the metacall used to initiate the task. This has the form:

```
exception(Type, Goal, Cont)
```

and indicates that an exception of type Type occurred when evaluating the goal Goal. The goal which caused the exception is replaced in the task by the variable Cont. This is a *metavariable*: it represents an as yet unspecified goal, to be executed in the place of the goal that caused the exception. Evaluation of the metavariable is suspended until such time as a program monitoring the task's status stream instantiates the metavariable to a term representing a new goal. As this component of the exception message is used to specify how evaluation should continue following an exception, it is referred to as a **continuation variable**. It is important to note that the generation of an exception message does not affect other processes in the task. These can continue executing.

Figure 4.2 illustrates the operation of the exception message by showing the sequence of events following a call to an undefined relation. It is assumed that a *monitoring process* is monitoring the status stream of an application *task* (represented, as usual, as a box). (a): an undefined relation goal is called in the task and, as a consequence, an exception message is generated to signal an exception of type undefined. (b): the erroneous goal is replaced in the task by the continuation variable Cont. (c): the monitoring program instantiates the continuation variable Cont to the Parlog primitive true. This instantiation passes the goal true to the task in which the exception occurred (by a form of back communication: Section 3.3.3). As the primitive true always succeeds, this has the effect of ignoring the exception.

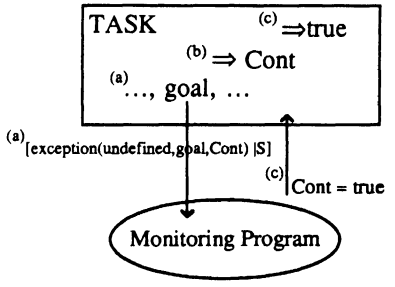


Figure 4.2 The exception message.

The application of the exception message and continuation variable is illustrated in

Section 4.4.1.

Parlog programs may generate exceptions explicitly using the raise_exception primitive. A call:

```
raise_exception(Type, Goal)
```

causes an exception of type Type involving the goal Goal to be signalled on the status stream of the task that made the call:

```
exception(Type, Goal, Cont)
```

The call to the raise_exception primitive is replaced by a continuation variable Cont, included in the exception message.

The generation of an exception and subsequent instantiation of the continuation variable has no logical reading. However, it has a simple *metalogical* reading. Exceptions generally represent goals that cannot be solved using the program with respect to which a query is being executed. For example, goals that call relations that are undefined in this program. A supervising program can evaluate such a goal: for example, it may be able to consult another program to find a definition for an undefined relation. When it instantiates the exception's continuation variable, it returns the result of its evaluation of the unsolvable goal. The supervising program thus extends the program used to execute the original query.

Parlog's exception handling mechanism has similarities to mechanisms found in other high-level languages. Ada, for example, permit exception handlers to be attached to program blocks [DoD, 1982]. In Multilisp, *catch* and *throw* primitives are used to bind exception handlers to a subcomputation, and to signal exceptions within the subcomputation, respectively [Halstead and Loaiza, 1985]. In Unix, exceptions are translated into signals, which may be trapped either by the process in which the exception occurred or by the process' parent. Parlog's mechanism is more powerful in some respects: as exceptions are implemented using messages rather than context switches, the task in which the exception occurred may continue executing while the exception is processed. The exception message and continuation variable provide a particularly elegant mechanism for signalling and responding to exceptions. Finally, as a Parlog exception handler can be implemented as a perpetual process that consumes a stream of messages, it can maintain an exception history as a local state (Section 3.3.6), without performing side-effecting operations.

4.2.3 Deadlock

Because of dataflow constraints, all processes in a Parlog task may be suspended waiting for data (Section 3.2.2). This is **deadlock** and should be regarded as an erroneous condition and reported. This is achieved by a further extension to the Parlog control metacall. The status message.

```
deadlock(N)
```

signals deadlock in a Parlog task. The message's argument N indicates the number of processes in the deadlocked task. Sections 6.2 and 6.3 describe applications of this status message.

For example, consider the program:

```
mode g(X?, Y↑), h(X↑, Y?).
g(a,b).          h(a,b).
```

A call to this program:

```
?- call( (g(X,Y), h(X,Y)), S, C).
```

results in the instantiation of the status variable S:

```
S = [deadlock(2) |_]
```

Neither g(X,Y) nor h(X,Y) can reduce because of dataflow constraints. g(X,Y) requires the value of X before it can produce Y; h(X,Y) requires the value of Y before it can produce X. This is deadlock. The argument to the deadlock message indicates that there are two processes in the deadlocked task.

A Parlog task may also contain no reducible processes because there is no producer for a particular data item within the task. For example:

```
?- call( consumer(X), S, C).
```

where consumer requires the value of X in order to reduce.

Such a task will be signalled as deadlocked, although it is not deadlocked in the classical sense; it is merely waiting for data from some external source. It does not appear feasible to distinguish between true deadlock and this 'apparent deadlock' in a Parlog implementation, so they are treated in the same way.

Data required by an 'apparently deadlocked' task can subsequently be produced by an external source. For example, a process producer(X) executing in another task may instantiate the variable X required by consumer. The task that was signalled as deadlocked hence becomes 'undeadlocked'. As Section 6.2 shows, it is useful to be able to detect this transition. It is hence signalled by a further status message:

undeadlock

4.2.4 Processor Scheduling

Any multiprogramming OS requires a mechanism for allocating processor resources to concurrently executing tasks. Typically, the OS maintains a list of executable tasks and invokes a **scheduler** to select a task to execute. Execution of a task proceeds until the task terminates or blocks or a timer interrupt occurs. The OS then reinvokes the scheduler to select a new task to execute.

The task created by the control metacall provides Parlog with a unit of computation to which processor resources can be allocated. In is thus useful to define a metacall control message priority(P) which can be used to associate a priority P with a task. Parlog system programs can use this control message to specify processor scheduling policies, by associating different priorities with different tasks. Task priorities are interpreted by the scheduler which actually selects tasks for reduction. The scheduler may be implemented in the kernel. Alternatively, as described in Section 5.4, it may be implemented in Parlog with the aid of simpler kernel mechanisms. This is somewhat less efficient but is more flexible and results in a simpler kernel.

4.2.5 Memory Management

Parlog's computational model assumes a kernel memory management mechanism that allocates memory for processes and terms as they are created and reclaims memory when these are no longer required. This mechanism must ensure that malicious or erroneous programs are not able to cause system failure by consuming all available memory.

This can be achieved by constraining the amount of memory that each task is able to consume. This limit can be specified either when the task is created using the control metacall or subsequently, by means of a control message.

An alternative, simpler (and therefore preferred) method is for the kernel to schedule an OS process when there is 'little free memory left'. If this process is provided with a term representing active tasks, it can abort unimportant tasks to free memory. Section 5.4 describes the implementation of a scheduler that can perform this function.

This mechanism can ensure that a Parlog OS cannot fail due to application program behaviour. It does not however permit OS or application programs to react to changes in the amount of free memory. This facility can be provided by a kernel-generated stream to which messages are appended periodically indicating how much free memory is available.

A Parlog OS kernel can implement more complex memory management mechanisms, such as virtual memory. Parlog system programs should be able to control the behaviour of such mechanisms by specifying policies (such as paging strategies). This issue is not considered herein.

4.2.6 Code Management

A basic resource management function that a Parlog OS must provide is management of the executable (or **object code**) form of OS and application programs. Code management involves a number of issues, including:

- manipulating code (for example, moving it between memory and disk)
- providing and controlling access to code
- permitting and controlling the modification of code while it is in use

As usual, these issues can be tackled either by extending the language or by providing simpler primitives that can be used to program mechanisms in the language. One approach to code mapping is implied by the Parlog control metacall call/3 (Section 3.4). This assumes a globally accessible dictionary structure, maintained by the kernel and accessed to retrieve the executable code for relations. Updates to this structure are presumably to be performed using side-effecting primitives. However, the semantics of concurrent accesses and updates to such a global structure are uncertain. Also, it appears difficult to maintain this structure consistent on multiprocessors.

A simpler scheme is therefore adopted, a modified version of a scheme described by Silverman *et al.* [1986], in the context of the related parallel logic programming language FCP. Two simple kernel mechanisms are provided that permit more complex

code management policies to be described in Parlog. The first is the representation of object code **modules** as Parlog terms. A module consists of executable versions of one or more procedures plus a dictionary data structure which can be used to access procedures defining particular relations. The dictionary associated with a module need not contain all of the module's relations. This permits certain relations to be hidden. The second is two kernel primitives. The first, Module@Goal, (usually written infix, as here) can be used to initiate execution of a goal Goal using the procedures in module Module. The second, 'CALL'(Goal), is a restricted form of the metacall call/1, that can be used to initiate execution of a Parlog *primitive* or of a procedure defined *in the same module* as the procedure by which it was called.

The representation of code as language terms means that code management can be programmed entirely in Parlog. The kernel is not required to maintain any centralized structures. This is illustrated in Section 4.3.1, where a code service is defined that caches terms representing modules and responds to requests for named modules. This code service can be instructed to replace a cached module with another. Subsequent requests are hence processed with respect to a new module; this implements run-time replacement of code.

These kernel mechanisms permit a new, *four-argument* control metacall to be defined in place of Parlog's 'global dictionary' metacall. A call to the four-argument metacall has the general form call(M,G,S,C) and initiates execution of a goal G using procedures in a module M. The application of this metacall is illustrated in subsequent sections. Its implementation is discussed in Chapter 5.

It is also possible to redefine the *three-argument* metacall. A call call(G,S,C) is interpreted as a call to a procedure G *in the same module* as the procedure in which it was called. Section 4.5.2 illustrates the application of this metacall.

4.3 Providing Services

An OS kernel provides OS programs with mechanisms that can be used to control hardware (Section 4.2.1). An OS must use these mechanisms to provide application programs with controlled access to resources such as terminals and disks. This can be achieved by defining terminal, disk etc. **services** to which application programs make requests. A service encapsulates its resource, protecting it from illicit access and sequencing accesses as required.

In imperative languages, a service is commonly implemented as a monitor (Section 2.2.2.2) or a process (Section 2.2.3). A service can be implemented in Parlog as a perpetual process which consumes a stream of terms representing requests. A Parlog service processes a request by performing some combination of the following five actions:

- spawning processes to perform computation;
- using primitives implemented in the kernel to side-effect hardware (this assumes a *procedure-call* based interface to devices: Section 4.2.1);
- generating requests to other services;
- binding variables in the request, to communicate values to the requesting task;
- recursing with altered internal state.

The order in which requests arrive on the service's input stream and dataflow constraints within the service define a total or partial temporal ordering on the processing of requests. Parlog's sequential conjunction operator can be used to sequence calls to side-effecting primitives.

Services can be characterized in terms of the actions they perform when processing requests. **Physical** services side-effect hardware by invoking kernel mechanisms; **logical** services do not. **Basic** services do not communicate with other services. A **filter** is a service which is composed with other services to augment their functionality. It accepts a stream of requests and passes on some different stream. A **mailbox** consumes a stream of requests and stream(s) of items (for example, an event stream: Section 4.2.1) and matches requests with items.

```
mode disk(DeviceAddress?, Requests?), cell(Contents?, Requests?).

disk(Device, [read(Addr, Block) |Rs]) ← read(Device, Addr, Block) & disk(Device, Rs). (C1)
disk(Device, [write(Addr, Block) |Rs]) ← write(Device, Addr, Block) & disk(Device, Rs). (C2)

cell(V, [read(V1) |Rs]) ← V = V1, cell(V, Rs).      % Return current contents.   (C3)
cell(V, [write(V1) |Rs]) ← cell(V1, Rs).            % Update contents.           (C4)
```

Program 4.1 Disk service and cell.

For example, 4.1 implements a basic physical service, disk, and a basic logical service, cell. disk implements a disk service. It processes a stream of read and write requests using side-effecting read and write primitives (C1,2). (The primitive read(D,A,B) has the logical reading: block B is located at address A in device D. write(D,A,B) has the same reading; it has the operational effect of writing block B to address A on device D). Note the use of the sequential conjunction operator ('&') in this procedure to sequence calls to these primitives. cell can be viewed as defining a memory cell. It responds to read and write requests by returning (C3) or updating (C4) its first argument, which represents its contents.

Services have a simple logical reading: they define a valid list of requests. This logical reading captures much of the functionality of a logical service. Physical services also have an important operational component, as they invoke side-effecting primitives.

Figure 4.3 illustrates a number of different services.

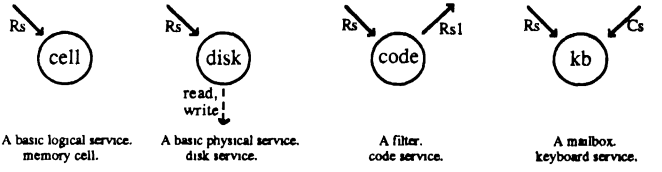


Figure 4.3 Types of service.

4.3.1 Filters: A Code Service

Program 4.2 implements a code service. This is a filter that accepts requests to read and write named modules and generates read and write requests to a disk service. This effectively implements a simple (non-hierarchical) file system. It may be composed with Program 4.1 above as follows:

```
..., code(Rs, Ds, N), disk(Device, Ds), ...
```

code(Rs,Ds,N) has the logical reading: Ds is the disk requests required to process requests Rs with respect to a cache of size N. It maintains a *file table* — a list of {module-name,disk-address} pairs — and a cache of retrieved modules: a list of {module-name,module} pairs. (Recall that a Parlog module is a language term: an executable code fragment: Section 4.2.6).

code/3 initially loads a file table from disk and primes the cache by loading the first N modules in this list. read_table(As,Ds,Ds1) (not defined) reads: the difference between the lists Ds and Ds1 is the disk requests required to load a file list As. prime_cache(N,As,C,Ds,Ds1) reads: the difference between the lists Ds and Ds1 is the disk requests required to load the first N modules listed in a file list As, and C is those N modules.

```
mode code(Requests?, Disk↑, Size?), code(Requests?, Addresses?, Cache?, Disk↑),
     read_table(Adds↑,Disk↑,Disk1?), prime_cache(Size?,Adds?,Cache↑,Disk↑,Disk1?),
     load(Adds?, Name?, Mod↑, Disk↑, D1?), save(Adds?, Name?, Mod?, Disk↑, D1?),
     lookup(Address?,Cache?,NewCache↑, Block↑), purge_oldest(Cache?,NewCache↑).

code(Rs, Ds, N) ←                                                        (C1)
     read_table(As, Ds, Ds1),                    % Load file table.
     prime_cache(N, As, Cache, Ds1, Ds2),        % Prime cache: N modules.
     code(Rs, As, Cache, Ds2).                   % Process requests.

code([code(Name, Module) |Rs], As, Cache, Ds) ←  % Request for a module:  (C2)
     lookup(Name, Cache, NewCache, Module1) :    % ... look in cache.
     Module = Module1,                           % ... return to requestor
     cache(Rs, As, [{Name, Module} |NewCache], Ds);  % ... add to front (note ';')
code([code(Name, Module) |Rs], As, Cache, Ds) ←  % Not in cache ...      (C3)
     load(As, Name, Module, Ds, Ds1);            % ... so load from disk.
     purge_oldest(Cache, NewCache),              % ... purge oldest entry
     cache(Rs, As, [{Name, Module} |NewCache], Ds1). % ... and add to cache.
code([module(Name, Module) |Rs], As, Cache, Ds) ←  % New module:        (C4)
     save(As, Name, Module, Ds, Ds1),            % ... save on disk.
     purge_oldest(Cache, NewCache),              % ... purge oldest entry
     cache(Rs, As, [{Name, Module} |NewCache], Ds1). % ... add to cache.

lookup(Addr, [{Addr, Block} |Cache], Cache, Block).   % Locate in cache; delete.  (C5)
lookup(Addr, [{Addr1, Block1} |Cache], [{Addr1, Block1} |NewCache], Block) ←     (C6)
     Addr1 =/= Addr : lookup(Addr, Cache, NewCache, Block).

purge_oldest([ B ], [ ]).                             % Delete oldest module.  (C7)
purge_oldest([B, C |Cache], [B |NewCache]) ← purge_oldest([C |Cache], NewCache).  (C8)
```

Program 4.2 Code service.

code/4 processes a request code(Name,Module) by either accessing the named module in the cache (C2) or, if it is not present (note the sequential clause search

operator, ';'), determining its disk address and generating requests to a disk service to load the module (C3). It processes a request module(Name,Module), which provides a new definition for a named module, by updating its cache and saving the new module on disk (C4). Subsequent requests code(Name,Module) are processed using this new module. This illustrates the replacement of code at run-time.

The cache is a fixed size and is maintained using a least recently used algorithm: each time a module is retrieved from the cache, it is deleted (C5) and inserted at the front (C2). When a new module is added to the cache (C3,4), the last module in the cache is deleted (C7).

Dataflow constraints mean that code processes requests sequentially: the lookup of the cache is performed in the guard, prior to the processing of further requests. It is however non-blocking: if a module must be requested from the disk service, cache can proceed to process further requests (C3). Subsequent requests for the same module do not require further disk accesses, unless it has been purged from the cache.

load and save(As,N,M,Ds,Ds1) (not defined) read: the difference between the lists Ds and Ds1 is the disk requests required to load/save a module M at the address A specified by the tuple {N,A} in the file list As.

4.3.3 Mailboxes: A Keyboard Service

Assume that the kernel generates a stream of Parlog terms representing characters typed at the keyboard. A simple keyboard service that matches these characters with requests for input can be implemented as:

```
mode kb1(Characters?, Requests?).

kb1([C | Cs], [R | Rs]) ← C = R, kb1(Cs, Rs).
```

kb1 takes as arguments a stream of keyboard input and a stream of variables representing requests for input. It suspends until both a character and a request are available and then 'receives' the character, returning it to the requesting process by back communication.

kb1 does not 'receive' characters typed until they are requested. It may be necessary to 'receive' characters — so as to check for interrupts, or to echo them on a terminal, for example — before they are requested. This can be achieved rather elegantly by maintaining two references to the input character stream. One is used to

eagerly receive and echo characters as they are typed; they other is used in a *lazy* fashion, to select characters as requests are received. The difference between the two references to the stream represents buffered characters: characters received but not yet requested.

Program 4.3 implements such a keyboard service. (Section 4.7 shows how this program is used in a Parlog OS). It can be called in a conjunction :

```
kb(Cs, Cs, Rs, Ts), screen(Ts), user(Rs)
```

The kernel is assumed to instantiate Cs to a stream containing both characters typed at the keyboard and a special constant break, which represents a user interrupt. This call provides kb with two identical references to this input stream Cs. screen is a service that displays characters on a screen; it accepts a stream containing characters to be displayed and the special constant cr, which requests a 'carriage return'. user generates a stream of requests Rs for characters.

```
mode kb(EagerInput?, LazyInput?, Requests?, Terminal↑).

kb([break |Cs], Cs1, Rs, [cr,b,r,e,a,k,cr |Ts]) ← kb(Cs, [break |Cs], Rs, Ts). % Break!   (C1)
kb([C |Cs], Cs1, Rs, [C |Ts]) ← C =/= break : kb(Cs, Cs1, Rs, Ts).          % Echo      (C2)
kb(Cs, [C |Cs1], [R |Rs], Ts) ← C = R, kb(Cs, Cs1, Rs, Ts).                 % Request   (C3)
```

Program 4.3 Keyboard service.

kb both matches application program requests for input with characters typed (C3) and scans ahead on the input stream, echoing characters typed by passing them to screen (C2) and processing interrupts (C1). It processes an interrupt, represented by the constant break, by passing the list [cr,b,r,e,a,k,cr] to screen. It also discards any buffered characters and adds the constant break to the buffer: this is achieved by recursing with the term [break|Cs] (not Cs1) as its second argument. The next application program request for input will thus receive the constant break.

kb ultimately reduces twice for each character typed: once to echo it, and once to communicate it to a requesting process. If characters are typed faster than they are requested, kb will tend to reduce more frequently using the second clause than the third. The first reference to the input stream (Cs) will thus point further down the stream than the second (Cs1). var tests can be used to establish priority between the

clauses. For example, adding a guard call var(Cs) to the third clause ensures that requests are not serviced if further input (characters or interrupts) is pending.

4.4 Communication with the Operating System

Application programs must be able to communicate with a Parlog OS to request access to services such as those described in Section 4.3. This can be achieved using Parlog's control metacall and the exception mechanism described in Section 4.2.2.

A Parlog OS uses the control metacall to execute an application program as a separate task. This protects the OS from application program failure. It also permits the execution of an application program to be monitored and controlled.

An OS process called a **supervisor** is associated with each such application task. This uses the metacall's status and control streams to monitor and control its task. The task is a user-level computation. The supervisor is part of the OS. (The distinction between user-level and OS tasks is conceptual, not physical: in a Parlog OS, both application and OS tasks can execute in the same address space; the lack of side-effecting primitives means that Parlog applications cannot compromise the correct execution of OS programs – though see Section 4.5).

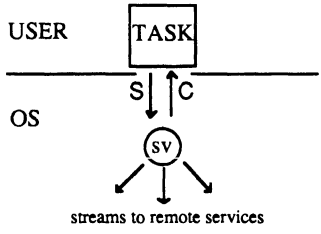


Figure 4.4 An application task and its supervisor (sv).

Figure 4.4 illustrates the relationship between a task and its supervisor.

Application program requests for OS services (**system calls**) are compiled into calls to the raise_exception primitive (Section 4.2.2). A distinctive exception type (say 'sys') identifies these exceptions as system calls. Thus a system call:

```
send(Service, Request)
```

when encountered in a Parlog application program is compiled as:

```
raise_exception(sys, send(Service, Request))
```

The compiler is assumed to recognize certain predefined predicates such as send as system calls and to compile them in this way.

System calls hence generate exception messages when executed at run-time. For example, a system call send(Service, Request) generates an exception:

```
exception(sys, send(Service, Request), Cont)
```

The monitoring supervisor process detects these exception messages and interprets them as requests for service. Two types of service are distinguished. **Local** services are implemented by the supervisor itself; an example of a local service is given below. **Remote** services are services such as those described in Section 4.3. Requests for remote services are translated into messages to the services concerned. Both the supervisor and remote services may communicate results back to a task that makes a system call by binding variables in the system call itself and/or by binding the continuation variable associated with the system call. This is the continuation variable incorporated in the exception message that notifies the task supervisor of the system call.

Parlog system calls are similar to supervisor calls in conventional operating systems in that they are trapped and handled by a supervisor program. In other respects, a Parlog system call is more akin to a non-blocking remote procedure call: it returns a result, but does not delay execution of other processes in the task that made the system call.

4.4.1 Local Services

The implementation of a simple local service is described. This serves to introduce task supervisors and to illustrate the use of local services.

A system call e_handler is implemented that permits an application program to register **exception handlers**. An exception handler is a procedure to be invoked in the application task when a particular type of exception occurs.

A call to e_handler has the general form:

```
e_handler(Type, Module, Relation)
```

It specifies a Relation (as defined in Module) that is to serve as the handler for

exceptions of a specified Type. For example, the call:

```
e_handler(undefined, M, undef_handler)
```

requests that the relation undef_handler (as defined in module M) be registered as the handler for exceptions of type undefined. This exception type indicates a call to an undefined relation.

Once an exception handler has been registered, goals in the application task that cause exceptions of the specified type are replaced by calls to the exception handler. The type of exception and the goal that caused the exception are provided as arguments to the exception handler call.

Program 4.4 implements a simple task supervisor that supports the system call e_handler as a local service. This supervisor takes as arguments a task's status and control streams and a list of exception handlers. It is initiated concurrently with the task that it supervises:

```
..., call(M1, G, S, C), supervisor(S, C, [ ]), ...
```

where G is the goal to be executed by the task and M1 is the module to be used to execute it. The third supervisor argument ([ ]) indicates that the set of handlers is initially empty.

The supervisor monitors its task's status stream (S). Recall that this may be instantiated to a stream of exception messages and terminated with constants succeeded, failed or stopped (Section 3.4). Calls to e_handler in application programs are assumed to be recognized by the compiler and compiled to calls to the raise_exception primitive in the same way as calls to send. Calls to e_handler in application programs hence generate exceptions when executed. The supervisor detects exceptions representing calls to e_handler and records exception handlers (C1). It replaces other exceptions with a call to a previously recorded exception handler, if one exists (C2) and halts execution if no exception handler can be found (C3). (The relation elookup(T,Hs,M,R) has the logical reading: the list Hs contains a tuple {T,M,R}). Note the use of the sequential clause search operator (';') to establish a default clause (C2). The procedure terminates when its task terminates (C4-6).

Note the use of the Parlog primitive =.. in clause C2. This is used to convert between lists and structured terms. A call to =.. suspends if both its arguments are variables. Otherwise it succeeds if its two arguments can be unified with a structured term $A_0(A_1,...,A_n)$ and a list $[A_0,A_1,...,A_n]$ respectively, and fails otherwise. =.. is used

here to construct a call R(T,G) from a relation name R and arguments T and G.

```
mode supervisor(Status?,Control↑,Handlers?), elookup(Type?,Handlers?,Mod↑,Relation↑).

supervisor ([exception(sys, e_handler(T,M, R), Cont) |S], C, Hs) ←          (C1)
        Cont = true,                        % e_handler system call succeeds.
        supervisor (S, C, [{T, M, R} |Hs]). %   Register error handler.
supervisor ([exception(T,G,Cont) |S], C, Hs) ← %                            (C2)
        T =/= sys,                          % Not a system call, so ...
        elookup(T, Hs, M, R)  :             %   Look for handler.
        Goal =.. [R,T,G],                   %   Found: construct handler call.
        Cont = M@Goal,                      %   Replace exception with call.
        supervisor (S, C, Hs);              %    (Note ';').
supervisor ([exception(T,G,Cont) |S], stop, Hs). % No handler: abort execution. (C3)
supervisor (succeeded, C, Hs).              % Termination ...                 (C4)
supervisor (failed, C, Hs).                 %                               (C5)
supervisor (stopped, C, Hs).                %                               (C6)
```

Program 4.4 Simple task supervisor.

Note also the use of the primitive @ (Section 4.2.6) in this clause. The call M@Goal, returned to the application task using the continuation variable Cont, initiates execution of Goal in module M.

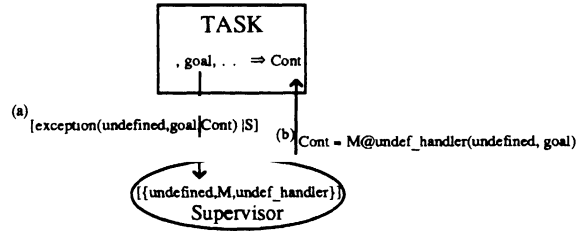


Figure 4.5 Implementation of the e_handler system call.

Figure 4.5 illustrates the execution of this supervisor. Assume that an application program has previously registered the error handler undef_handler using a system call e_handler(undefined,M,undef_handler). The exception handler has hence recorded the tuple {undefined,M,undef_handler}. A call to an undefined relation goal then occurs in the application task. An exception message is generated and is intercepted by the task's supervisor (a). This looks up its list of exception handlers and uses the

exception message's continuation variable to return a call to the error handler undef_handler to the task (b).

4.4.2 RPC and Circuit Access to Remote Services

A supervisor provides application programs with access to services such as those described in Section 4.3 by translating system calls into messages to such services. Two types of access can be provided, RPC and circuit. **RPC** (Remote Procedure Call) access requires the application program to make a system call each time it wishes to access a particular service. A generic RPC system call has the form:

```
send(Service, Request)
```

A task supervisor that receives an exception message representing such a system call translates it into a message to the service named Service. The message has the form:

```
{Request, Cont, Term}
```

where Cont is the system call's continuation variable and Term is the termination variable of the application program task that made the system call. A **termination variable** is a unique variable associated with a task by its supervisor. The supervisor guarantees to instantiate this variable to some non-variable term when the task terminates. This termination variable permits services to which a request is sent to detect if and when the task that generated the request terminates: if a var test indicates that a termination variable is uninstantiated, then the task has not terminated. The generation of termination variables is illustrated in Section 4.7.

Circuit access requires the application program to make an initial system call to request a stream or *circuit* to a service. Requests to the service can subsequently be appended directly to that stream. A generic circuit request has the form:

```
send(Service, circuit(Stream))
```

This is translated by the task supervisor into a request:

```
{circuit(Stream), Cont, Term}
```

to Service.

The application program may instantiate the variable Stream to a stream of requests:

```
Stream = [Request1, Request2, ... , RequestN]
```

These are processed by the service that receives them as if the application program had made a sequence of system calls:

```
..., send(Service, Request1) , send(Service, Request2), ...
```

The ordering of requests on a circuit stream determines the order in which they are received by the service.

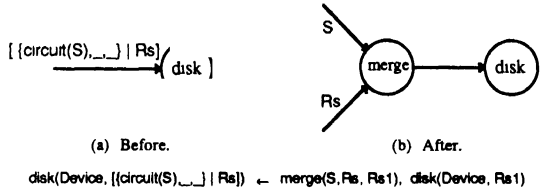


Figure 4.6 Circuit access to a disk service.

A service may process a request for a circuit in two ways. It may merge the circuit stream in with its usual request stream. For example, Program 4.1 may be augmented with an additional clause to handle requests for circuits. The additional input stream (S) is merged with the original input stream, as illustrated in Figure 4.6. Recall that the merge relation (Program 3.3) produces on its output argument a stream that is some intermingling of its two input arguments.

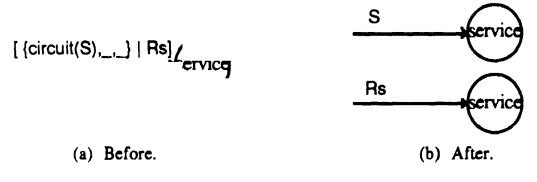


Figure 4.7 Circuit access to a stateless service.

Alternatively, if the service is stateless and there is hence no need to sequence accesses, a copy of the service may be created to process requests on the circuit, as illustrated in Figure 4.7. This avoids any overhead associated with the merge process.

4.4.3 Termination and Errors

A service that receives a system call request from an application task may need to:

- pass results back to that task
- notify the task of erroneous conditions detected while processing the call
- notify the task that processing of the call has terminated.

A service can pass results to a task by back communication (Section 3.3.3): that is, by binding variables in a request. For example, the disk service (Program 4.1) uses back communication to return a disk block following a read request. A variable used in this way (such as the variable Block in the read request) serves as an implicit return address.

Back communication can also be used to notify a task of errors detected by a service when processing a system call. (For example, the Block variable in Program 4.1's read request could be bound to a special constant error). However, the task then has the option of ignoring the error condition. A better method is to use a system call's continuation variable for this purpose. Recall that a system call is translated into an exception and hence is represented in the application task as a continuation variable (Cont). As noted in Section 4.4.2, this continuation variable is included in the request message generated by the task's supervisor following the system call. A service receiving this request message can then bind the continuation variable to a call to the raise_exception primitive:

```
Cont = raise_exception(Type, Goal)
```

where Type is the type of error the service wishes to signal in the application task and Goal is the original system call.

This back communication of a new goal generates a further exception in the application task. The task's supervisor thus receives a further exception message, indicating why the system call could not be processed. It can then choose to process the call elsewhere, to ignore it, to abort the application, etc. This use of the continuation variable to communicate goals between tasks is illustrated in Section 4.5.2. It demonstrates the power of the logical variable and in particular of the exception message's continuation variable.

This error-reporting mechanism cannot be used directly in the case of circuit access, as a circuit is established by a single system call. Only one continuation variable is

available for all requests made on the circuit. A filter process must therefore be interposed between the task making requests on the circuit and the service that receives them. This associates a new continuation variable with each request received on the circuit and uses these new variables to monitor the progress of the requests. Any errors are signalled to the application task by binding the original continuation variable.

This use of a filter process is illustrated in Figure 4.8. In (a), a circuit request {circuit(S),T,C} is received by a service. In (b), a filter has been created. This accepts a stream of requests R1, R2, etc. and forwards a stream of tuples {R1,T,C1}, {R2,T,C2}, etc. to the service. The filter monitors the new variables C1, C2, etc.; if any is bound to an exception message by the service, this is signalled in the application task by instantiating the original continuation variable, C.

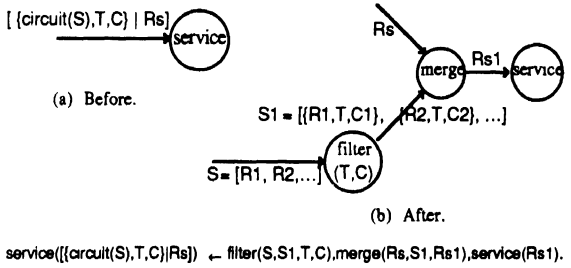


Figure 4.8 Filtering a circuit.

The continuation variable associated with system calls also permits a service to notify a task that processing of a system call has terminated. A system call in an application program is replaced by a continuation variable when executed and hence suspends until the continuation variable is bound to a new goal. The continuation variable is passed to the service addressed by the system call. This can either:

- instantiate the continuation variable (to true) *after* processing the system call, or
- validate the system call (Section 4.5.2) and then instantiate the continuation variable (to true) *before* processing the system call.

In the former case, the system call does not terminate until the service has processed the request. In the latter case, the system call terminates immediately the call is validated; this corresponds to a spooling of the request.

4.4.4 Discussion

It has been proposed that application program access to OS services be provided by system calls. These are implemented using the metacall exception feature introduced in Section 4.2.2. Exceptions representing system calls are translated by a task supervisor into requests to OS services. The continuation variable associated with a system call is used to notify a task's supervisor of errors detected during its processing. This use of the continuation variable for error notification provides automatic 'routing' of error messages to a task's supervisor. It also ensures that an application program cannot ignore errors.

Two types of access to remote services have been described: RPC and circuit. Each has its advantages and disadvantages.

Sequencing: the stream associated with circuit access can be used to explicitly sequence requests. RPCs can only be sequenced using sequential operators or by using the results of requests as synchronization tokens to delay the generation (or processing) of subsequent requests.

Performance: Circuit access has set-up and shutdown costs, if a filter must be created; RPC does not. However, circuit access does not require routing of requests once a circuit is established. RPC access requires that each request be interpreted and routed.

Bottlenecks: in a multiprocessor, a circuit may be a bottleneck, particularly if requests do not need to be serialized. In contrast, RPC access permits system calls to be handled and routed independently at each node on which the application task is executing (see Section 6.5). This also permits requests for services replicated on several nodes to be routed to a node that is 'nearby'.

Termination and Errors: RPC access associates a continuation variable with each request. This can be used to report termination and errors. A special filter process is required to provide similar functionality when using circuit access.

Validation: Circuit access requires additional validation (see Section 4.5).

It is suggested that RPC access be used when services are accessed rarely, or are replicated on many nodes. Circuit access is to be preferred when many requests must be made to the same service, particularly if these requests need to be sequenced.

4.5 Robustness

An OS service is robust if no application program behaviour can cause it to execute erroneously. This section discusses how Parlog OS services such as these described in Section 4.3 can be made robust.

A Parlog process can affect the execution of another Parlog process in two ways: by failure and by the unification of shared variables. (Side-effecting primitives could also be used to modify a process' data state. However, although side-effecting primitives are required to control hardware, their use is assumed to be isolated inside specialized system programs). Parlog's control metacall can be used to encapsulate processes so as to localize failure and run-time errors. The chief problem to be resolved when seeking to construct robust system services is thus that of shared variables.

Application programs and system services come to share variables as a result of system calls, which cause a message containing terms generated by the application program to be passed to a system service. The only way that an application program can compromise the correct execution of a system service is thus by generating an invalid system call.

A system call generally consists of a name and a set of arguments. Arguments have output and input components. *Output* components are ground terms to be passed *to* the service; *input* components are variables used to receive values *from* the service. For example, a system call read(Addr,Block) made to a disk service (Program 4.1) has name read, output component Addr and input component Block. The processing of a system call consists of two phases: *match* and *reply*. In the match phase, the service accesses the name and output components and computes a reply (if any). In the reply phase, back communication occurs as any reply is unified with the input components of the call.

An invalid call can cause a service to behave erroneously in both phases. In the match phase, the service can *suspend* because the name or an output component is a variable. This is a **suspension** error: a service should never suspend waiting for application program data, as it cannot be known when that data will become available. A service can also *fail* in the match phase, because the name or an output component has an invalid value. This is a **matching** error. In the reply phase, a service may fail because an input component that it attempts to instantiate is already bound to another value. This is a **unification** error.

For example, consider Program 4.1. Table 4.1 lists five potential causes of erroneous behaviour due to an invalid RPC system call. (Similar problems are associated with circuit access; these are not considered here). It is assumed that the disk service expects calls of the form read(Addr,Block), where Addr is an integer and Block is a variable, or write(Addr,Block), where Addr is an integer and Block is a valid block. The example shows the request component of an erroneous system call.

#	Description	Example	Error	Result
1.	*incorrect* value	read(abc, Y)	1st argument not integer	service fails
2.	*missing* value	read(X,Y)	1st argument not bound	service suspends
3.	*unexpected* value	read(10, abc)	2nd argument not variable	service fails
4.	*incorrect* call	foo(X)	only read or write valid	service fails
5.	*missing* call	X	call should be bound	service suspends

Table 4.1 Potential causes of disk service failure.

Errors 1 and 4 are matching errors. Errors 2 and 5 are suspension errors. Error 3 is a unification error.

Simple validation techniques permit these errors to be avoided. Suspension errors are avoided by not forwarding *partially constructed* system calls (that is, those in which the name or output components are not ground) to the service. Matching errors are avoided by either validating system calls before forwarding them to the service or by extending the service with code that detects and accepts invalid formats.

Unification errors are avoided by:

(1) Replacing input components in a request with new variables before processing the request.

(2) Creating 'lazy copy' processes to unify these new variables with the original input variables once the new variables are bound by the service. This ensures that the service never has direct contact with application program input variables.

(3) Executing copy processes either as part of the application program or encapsulated in a separate control metacall. This ensures that a unification error does not cause failure of the service.

A *generic* copy procedure, that can be used if a service's output consists only of ground terms, may be defined as follows:

```
mode copy(ServiceOutput?, ApplicationInput↑).

copy(T, T) ← atomic(T) : true.                                    (C1)
copy([H1 |T1], [H |T]) ← copy(H1, H), copy(T1, T);                (C2)
copy(S1, S) ← S1 =. L1 : copy(L1, L), S =. L.                     (C3)
```

A call to this procedure suspends until its first argument is instantiated. Constants are then copied from input to output (C1). (The primitive atomic suspends until its argument is bound and then succeeds if it is a constant and fails otherwise). Lists are copied by copying their head and their tail (C2). Other structures are converted to lists and then copied (C3).

A more complex copy procedure is required if service output incorporates variables that are to be passed to the application program. This copy procedure must be aware of the structure of the input component being copied so that it can pass variable subcomponents immediately instead of waiting for them to be instantiated.

Validation techniques that avoid suspension, matching and unification errors can be applied in the application programs themselves (**at source**), in services (**at destination**) or in between (**en-route**). These three approaches are described and illustrated with examples.

4.5.1 At-source

Compile-time transformation of application programs can be used to ensure that only valid system calls are generated and hence that only valid requests reach a service. For example, a call:

```
send(disk, read(A, B))
```

can be compiled to a call to the procedure diskread:

```
diskread(A,B) ← integer(A) : raise_exception(sys, send(disk, read(A, B1))), copy(B1,B).
```

The integer primitive suspends until its argument is instantiated and then succeeds if it is an integer, and fails otherwise. This avoids *suspension* and *matching* errors (Errors 1 and 2 in Table 4.1). The copy process (described above) makes the value of

the new output variable B1 available as it is generated by lazily unifying it with B. If the call to copy fails because B is already bound to a different value, the application program rather than the OS service fails. This avoids *unification* errors (Error 3).

Errors 4 and 5 cannot occur in this example as the arguments to the system call send are known at compile-time. This approach is not practical if system call arguments can be generated at run-time.

4.5.2 At-destination

An alternative approach is to make the service itself robust. *Matching* errors are dealt with by extending the service with extra code that validates call arguments. *Unification* errors are dealt with using a copy process as before. This is executed locally, encapsulated in a control metacall. *Suspension* errors are best dealt with by introducing a filter that delays requests until they are sufficiently constructed to be processed. This is an example of en-route validation and is discussed in the next section.

```
mode disk(Device?, Requests?), try(X?, Y?, Cont↑, Goal?), result(Status?, Cont↑,Goal?),
     valid_address(Address?), valid_block(Block?).

disk(De, [{read(Addr, Block), Term, Cont} |Rs]) ←                          (C1)
     valid_address(Addr) :          % Check that address is valid.
     read(De, Addr, B1) &           % Read block using primitive.
     try(B1, Block, Cont, read(Addr, Block)),   % Perform unification.
     disk( De, Rs).
disk(De, [{write(Addr, Block), Term, Cont} |Rs]) ←                         (C2)
     valid_address(Addr), valid_block(Block) :  % Check that arguments are valid.
     Cont = true &                  % Succeed call.
     write(De, Addr, Block) &       % Write block using primitive.
     disk(De, Rs);                  % Note ';'.
disk(De, [{Other, Term, Cont} |Rs]) ←   % Otherwise signal exception.    (C3)
     Cont = raise_exception(badarg, send(disk, Other) ), disk(De, Rs).

try(B1, B, Cont, O) ← call(copy(B1, B), S, C), result(S, Cont, O).         (C4)

result(succeeded, true, O).                    % Success: Cont = true.    (C5)
result(failed, raise_exception(badarg, send(disk, O)), O).  % Unification failed: error. (C6)
```

Program 4.5 Robust disk service.

Program 4.5 implements a robust file service. It is assumed here that messages generated by task supervisors have the form {Request, Term, Cont}, where Term is a termination variable and Cont is the system call's continuation variable. As this tuple is generated by the task supervisor, an OS process, the service only needs to validate the Request component.

valid_address(A) (not defined) has the logical reading: A is a valid address. valid_block(B) reads: B is a valid block.

Program 4.5 copes with matching errors (Errors 1 and 4) by testing that arguments are valid (C1,2) and including a default clause to deal with invalid requests (C3). The default clause signals invalid requests by instantiating the continuation variable Cont to a call to the raise_exception primitive. This causes an exception message:

```
exception(badarg,send(Disk,O),NewCont)
```

to be signalled on the status stream of the task that made the request. badarg is the type of exception. send(Disk,O) is the system call which caused the exception. NewCont is a new continuation variable, which the task's supervisor can use to respond to the exception.

Unification errors (Error 3) are avoided by calling the relation try/4 (C1), which uses the three-argument form of the control metacall (Section 4.2.6) to program **fail-safe unification** (C4). The call:

```
call(copy(Block1, Block), S, C)
```

attempts to lazily unify terms Block and Block1. The use of the control metacall ensures that this always succeeds, even if unification fails. Examining the status variable S indicates the result of the unification operation: succeeded or failed (C5,6). This is signalled to the application program by instantiating the system call's continuation variable to either true (to indicate success: C5) or to a call to the raise_exception primitive (C6).

Note that *in this example* the use of the control metacall and copy process is not in fact necessary. As the read primitive returns immediately with a block B1, rather than constructing it incrementally, clause C1 can instead be written:

```
disk(De, [{read(Addr, Block), Term, Cont} |Rs]) ←                    (C1)
      valid_address(Addr) :                  % Check that address is valid.
      read(De, Addr, B1) &                   % Read block using primitive.
      Cont = (Block = B1) &                  % Return unification operation
        disk(De, Rs).
```

The system call's continuation variable is used to return the unification operation Block=B1 to the application task. Failure of the unification operation hence causes failure of the application task rather than the service.

4.5.3 En-route

Finally, requests can be validated after they are generated by the application program and before they are received by the service. A simple example of en-route validation is a filter that delays incomplete requests. Program 4.6 implements such a filter. This can be composed with Program 4.5 to avoid suspension errors (Errors 2 and 5). It delays requests until *either* they are sufficiently instantiated *or* the task that generated them has terminated. In the former case it forwards them to the service; in the latter, it discards them. (It is assumed that system calls made by a task that has failed or been aborted are not to be processed).

```
mode delay_filter(Requests?, RequestsOut↑), delay(MessageIn?, Term?, MessageOut↑),
      sufficiently_ground(Message?).

delay_filter([ {Request, Term, Cont} | Rs1], Os) ←     % Create delay process.   (C1)
      delay({Request, Cont}, Term, Os2), merge(Os1, Os2, Os), delay_filter(Rs, Os1).

delay(Message, Term, [Message]) ← sufficiently_ground(Message) : true.          (C2)
delay(Message, Term, [ ]) ← data(Term) : true.     % Task termination: discard. (C3)
```

Program 4.6 Delay filter.

Program 4.6 creates a delay process for each request received. As noted in Section 4.5.2, requests have the form {Request,Term,Cont}. merge processes combine the output of the delay processes and forward it to the service (C1). Each delay process performs two checks concurrently. It checks whether its request is sufficiently instantiated, using the procedure sufficiently_ground; if it is, delay outputs it and

hence forwards it to the service, minus its termination variable (C2). (sufficiently_ground(M) reads: M is a valid message). It also checks whether the task that made the request has terminated, by examining its termination variable, Term. If it is, the request is discarded (C3). Both tests use Parlog's data primitive to determine whether values are available. A delay process hence suspends until either the request is sufficiently bound or the termination variable is instantiated.

The procedure sufficiently_ground can easily be extended to check for matching errors. Unification errors can also be dealt with by a filter of this sort.

4.5.4 Discussion

The three approaches to error avoidance discussed above all apply the same basic techniques to avoid suspension, matching and unification errors and hence to obtain robust services. They each have advantages and disadvantages.

At-source avoidance isolates errors within application programs. OS programs can thus be simpler. No filters or termination variables are required to isolate and terminate incomplete requests. In contrast, at-destination validation requires potentially complex mechanisms for dealing with partially constructed requests.

On the other hand, at-source validation requires that the compiler (perhaps an application program) be trusted. The compiler must be aware of all services' request formats; this implies a potentially undesirable degree of global knowledge in a distributed system. Also, it is difficult to deal with requests constructed at run-time in this way.

En-route validation is conceptually the simplest, as it separates validation of requests from both the generation and processing of system calls. Also, the services themselves then need perform no error checking. This means that trusted OS components can be permitted to make requests to services directly, thus incurring no overhead. The only disadvantage of en-route validation is the potential overhead of processing request streams twice: once en-route, to validate requests, and once at the service itself.

For most purposes, en-route validation is the simplest and most effective approach.

4.6 Naming

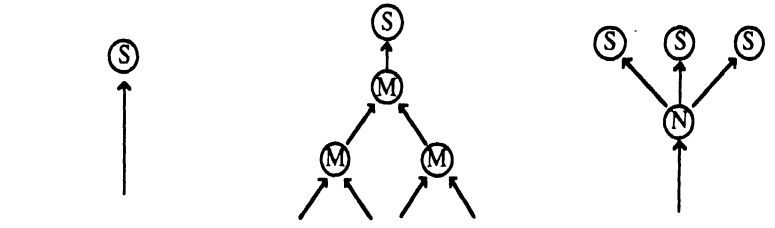
Application programs use system calls to refer to services to name (Section 4.4). Parlog provides no direct support for the naming of processes or other computational objects. However, the logical variable provides **global reference**: a process possessing a reference to a logical variable can access that variable, wherever it is located. A Parlog OS can use logical variables to implement a variety of naming schemes.

The request streams used to access Parlog OS services are logical variables. Logical variables can thus be regarded as naming **ports** that provide access to services. A logical variable can only be accessed by processes that possess a reference to it, and references cannot be forged (they can only be copied or equated by unification). Logical variables can thus also be viewed as unforgeable **capabilities** [Dennis and van Horn, 1966] for services. A process must possess a reference to a service's request stream before it can access that service. Logical variables can also be compared with mail addresses in actor systems [Agha, 1986]. Like actor addresses, but unlike certain implementations of capabilities, logical variables can be incorporated in data structures and messages and tested for identity (using Parlog's equality primitive, ==).

The single-assignment property of Parlog's logical variable effectively means that only a single process can be connected to a port. However, merge processes (Program 3.3) can be used to multiplex several connections. In principle, this use of merge incurs significant overhead: a binary tree of N - 1 merge processes multiplexing N connections requires $O(log_2 N)$ reductions to forward a single request. However, simple optimizations in an implementation can reduce this overhead to a small constant [Shapiro and Safra, 1986].

Symbolic naming schemes can be programmed in Parlog using **name servers**: processes that have direct connections to services and which forward requests tagged with symbolic names to these services. This corresponds to late binding of names to services. Early binding is also possible: a program which is known to require access to a particular service can be given a direct stream connection to that service at compile time.

Figure 4.9 illustrates a service and its request stream or port; a tree of merge processes, used to multiplex several connections to a single service; and a name server. In this figure, S denotes a service, M a merge process and N a name server.



(a) A service and its port. (b) Several connections multiplexed to a service. (c) A name server.

Figure 4.9 Services, ports and name servers.

Program 4.7 implements a name server. This has as arguments a stream of requests, tagged with symbolic names, and a list of {Name,Stream} pairs. This list defines the naming relation applied by the name server to names in incoming requests.

The name server processes requests received in the form {Destination, Message, Term, Cont} by looking up its list of names for Destination (C2,3). If it knows of such a service, it forwards the request in the form {Message, Term, Cont} (C2). Otherwise, it signals an exception of type unknown_service, using the request's continuation variable Cont (C4).

```
mode names(Requests?, Names?), send(To?, Message?, Term?, Cont?, Names?, Names↑).

names([{To, M, T, Cont} | Rs], Ns) ← send(To, M, T, Cont, Ns, Ns1), names(Rs, Ns1).   (C1)

send(To, M, T, Cont, [{To, St} |Ns], [{To, St1} |Ns]) ← St = [{M, T, Cont} |St1].      (C2)
send(To, M, T, C, [{N, St} |Ns], [{N, St} |Ns1]) ← To =/= N . send(To, M, T, C, Ns, Ns1).   (C3)
send(To, M, Cont, [ ], [ ]) ←                                                          (C4)
    Cont = raise_exception(unknown_service, send(To, M) ).
```

Program 4.7 Name server.

The code service implemented in Program 4.2 is also a type of name server. This maps requests for named modules to modules.

4.7 A Parlog Operating System

This section links together OS components introduced in previous sections to define a simple Parlog uniprocessor OS. This provides a context for their use. It also demonstrates that the solutions to OS design problems presented in this chapter constitute a coherent methodology for OS design and implementation.

Multiprocessor execution of this simple OS is considered in Chapter 6.

4.7.1 Operating System

The simple OS supports two system calls: send/2 and task/4. send/2 provides RPC and circuit access to four basic services: screen, kb, disk and code. task/4 requests the creation of a new user-level task. A task created using task/4 has its system calls trapped and processed by the underlying OS.

The screen service processes requests to display characters at a terminal. The keyboard service (kb) processes requests for characters typed at a keyboard. It also echoes characters to the screen service as they are typed. The disk service processes requests to read and write disk blocks. The code service maintains a table of {module-name, disk-address} pairs and translates requests for named modules into the read and write requests to disk required to load them.

```
mode os_init(KeyboardInput?, InitialModule?, InitialGoal?).

os_init(Is, Name, Goal) ←
        call(M, Goal, Si, [priority(2) |_]),          % User-level task.
        sv(Si, So, Term, Ns),                         % Task supervisor.
        names(Ns, [{kb, Ks}, {screen, Ss2}, {disk, Ds2}, {code, Cs}]),  % Name server.
        kbfilter(Ks, FKs), screenfilter(Ss2, FSs2),   % Filters for
        diskfilter(Ds2, FDs2), codefilter(Cs, FCs),   %   validation.
        kb(Is, Is, FKs, Ss1), screen(Ss),             % kb + screen.
        code(10, [code(Name, M) |FCs], Ds1), disk(Ds),  % code + disk.
        merge(Ss1, FSs2, Ss), merge(Ds1, FDs2, Ds).
```

Program 4.8 Operating system bootstrap.

The OS is initiated by a call to the procedure os_init (Program 4.8). Figure 4.10 illustrates the process network created by this procedure.

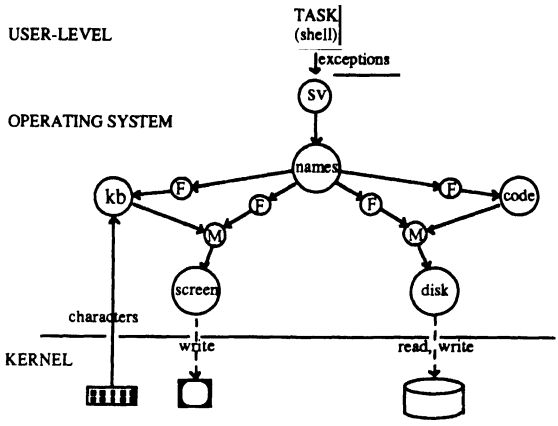


Figure 4.10 Parlog operating system.

A call to os_init might have the form:

```
os_init(ls, shell, shell)
```

The first argument, ls, is a stream of characters generated by the kernel, representing characters typed at the keyboard. This implements a unification-based interface to the keyboard service (kb: Program 4.3). The second and third arguments indicate the module and goal to be executed by an initial user-level task. To initiate this task, a message code(Name,M) is sent to the code service (Program 4.2) to retrieve the named module from disk. Execution of the initial goal using this retrieved module is then initiated using the four-argument control metacall. A task supervisor (sv: Program 4.9; described below) is created to monitor execution of the new task. This traps send system calls and forwards RPC and circuit requests to the name server (names: Program 4.7), which routes them to services. The various filter processes (kbfilter, etc; represented as F in Figure 4.10: not defined, but see Section 4.5.3) perform en-route validation of requests. They accept a stream of requests augmented with termination and continuation variables and pass on a stream of validated requests.

screen (not defined) and disk (Program 4.1) are assumed to use side-effecting primitives to implement procedure call-based interfaces to their devices. merge processes (represented as M in Figure 4.10: Program 3.3) are used to combine requests made directly to the disk with disk requests generated by the code service and to combine requests made directly to the screen with characters echoed by the keyboard service.

4.7.2 Task Supervisor

The task supervisor (sv: Program 4.9) processes messages received on a user-level task's status stream (Si) and generates a filtered status stream, minus system calls (So). (So is ignored in the case of the initial application task, but permits further application tasks created using task/4 to be monitored by the application task that initiated them). It also takes as arguments a termination variable, to be bound when the task terminates (Term) and a request stream to the name server (Ns).

```
mode sv(StatusIn?, StatusOut↑, TermVar↑, NameServerRequests↑).

sv([exception(sys, send(To,Msg), Cont) |Si], So, Term, [{To, Msg, Term, Cont} |Ns]) ← (C1)
        sv(Si, So, Term, Ns).                    % send system call: forward.
sv([exception(sys, task(Mo,G,So1,C), Cont) |Si], So, Term, Ns) ← (C2)
        call(Mo, G, Si1, [priority(1) |C]),      % task system call: create task
        merge(Ns1, Ns2, Ns),                     % Merge name server requests.
        sv(Si1, So1, Cont, Ns2),                 % Create new task supervisor.
        sv(Si, So, Term, Ns1);                   % (Note ';')
sv([Other |Si], [Other |So], Term, Ns) ← sv(Si, So, Term, Ns). % Forward other status. (C3)
sv(succeeded, succeeded, true, [ ]).             % Termination ...          (C4)
sv(failed, failed, true, [ ]).                   %                         (C5)
sv(stopped, stopped, true, [ ]).                 %                         (C6)
```

Program 4.9 Task supervisor.

send/2 system calls are augmented with a termination variable Term and the system call's continuation variable Cont and forwarded to the name server (C1).

task/4 system calls result in the spawning of a new supervisor and a new task using the control metacall (C2). Status messages that do not represent system calls are forwarded on the filtered status stream (C3). Termination is handled by closing the name server request stream and instantiating the termination variable (C4-6). (Recall that upon termination, a metacall's status stream is terminated with a constant: Section 3.4). The continuation variable of the task/4 system call is retained by the new task supervisor as a termination variable. The supervisor sv instantiates this termination variable, to true, upon termination; this means that a task/4 system call succeeds only when the task it has created terminates.

Figure 4.11 illustrates the creation of a new task (C2), showing the process network before and after the processing of this system call. Note how :

- the new task's status stream Si1 is filtered (by the new sv process) to remove system calls before being passed to its parent task as So1.
- the new supervisor's stream of requests to the name server is merged with that of the parent supervisor.

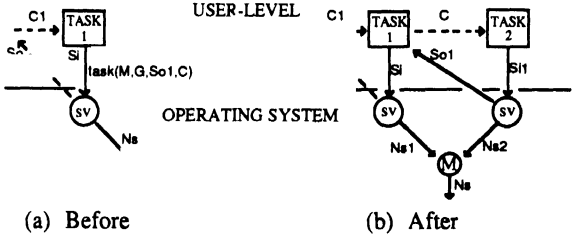


Figure 4.11 Task supervisor: creation of a new task.

4.7.3 User-level Program

The initial user-level task created when the OS is initialized is able to:

- request a keyboard service for characters typed at the keyboard;
- request a screen service to display characters on the screen;
- request a disk service to read and write disk blocks;
- request a code service for named modules;
- create a new user-level task with the same capabilities, which it is able to monitor and control.

The initial user-level task can most usefully define a shell: a program that reads, interprets and executes user queries. Program 4.10 gives a possible definition for such a user-level shell program. This executes a series of queries typed at the keyboard, each as a separate task.

shell first makes send/2 system calls to establish circuit connections to the keyboard and code services, Ks and Cs respectively (C1). read(Ks,Rs) (not defined) has the logical reading: Ks is a list of characters typed at the keyboard and Rs is the list of queries these characters represent. read/2 requests characters from the keyboard using its circuit (Ks) and constructs terms representing queries, which it passes to shell/3.

```
mode shell, shell(Requests?, Code↑, Synch?), monitor(Status?, Control↑, Synch↑),
     read(Keyboard↑, Requests↑).

shell ←                                                              (C1)
    send(kb, circuit(Ks)),                    % Obtain circuit to keyboard.
    send(code, circuit(Cs)),                  % Obtain circuit to code.
    read(Ks, Rs),                             % Read queries ...
    shell(Rs, Cs, true).                      %   ... and process queries.

shell([fg(Name, Goal) |Rs], [code(Name, M) |Cs], true) ← % 'foreground': get module (C2)
    task(M, Goal, S, C),                      %   ... and create task.
    monitor(S, C, Synch),                     %   monitor task's execution.
    shell(Rs, Cs, Synch).                     %   Synch delays next query.
shell([bg(Name, Goal) |Rs], [code(Name, M) |Cs], true) ← % 'background'  (C3)
    task(M, Goal, S, C),
    monitor(S, C, _),
    shell(Rs, Cs, true).                      %   Synch = true: don't delay.

monitor([exception(_,_,_) |_], stop, true).   % Exception: halt task.   (C4)
monitor(succeeded, _, true).                  % Termination.            (C5)
monitor(failed, _, true).                                             (C6)
```

Program 4.10 User-level shell.

Figure 4.12 shows the process network created by a call to shell.

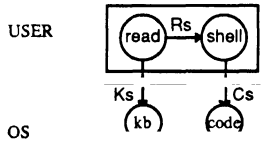


Figure 4.12 Shell process network.

Queries are assumed to have the form fg(Name,Goal)or bg(Name,Goal). shell/3 uses the task/4 system call to execute these queries (C2,3). It uses its circuit to the code service (Cs) to retrieve code as required.

A monitor/3 process is created for each query. This monitors the execution of the task, aborting its execution of exceptions occur (C4) and binding a synchronization variable Synch when it terminates (C4-6). The synchronization variable is used in the

case of a foreground query (C2) to delay the creation of further tasks until the query has terminated.

4.7.4 Processor Scheduling

The simple OS presented here also illustrates the use of the control metacall's priority(_) control message (Section 4.2.4) to specify scheduling policies. Assume that three priority levels, 1-3, are recognized by the kernel's scheduler, where 3 is the highest priority. Assume also that the OS is initiated with priority 3:

```
call(os_init(ls,shell,shell), S, [priority(3) |C])
```

The initial user-level task (the shell) has been scheduled at priority 2 (Program 4.8):

```
call(M, Goal, Si, [priority(2) |_])
```

while the task supervisor schedules subsequent tasks at priority 1 (Program 4.9: C2):

```
call(Mo, G, Si1, [priority(1) |C]),
```

The OS thus informs the underlying scheduler that OS processes are to be allocated processor resources at a higher priority than the initial application task, the shell, which in turn has a higher priority than other application tasks. Just how these priorities are interpreted depends on the OS kernel's implementation of processor scheduling (see Section 5.4).

Note that the task supervisor presented here does not prevent the shell from increasing the priority of application tasks, using a priority(N) message with N > 1. A small extension to the task supervisor is required to prevent this: this filters application tasks' control streams and removes such priority messages.

4.8 Abstractions in User-Level Programs

Application (user-level) programs need to be able to construct abstractions based on the simple services provided by an OS such as that presented in Section 4.7. For example, rather than transmitting terminal control codes to a screen service, application programs may wish to perform operations on 'windows'. 'window' is an abstract data type (ADT) on which operations such as 'create', 'delete', etc. must be defined.

This section looks briefly at three techniques for the implementation of abstractions in Parlog. The application of the latter two techniques in a user-level program is illustrated in Section 7.4.

4.8.1 Procedures

Operations on ADTs can be defined as procedures which themselves use system calls to access OS services. For example, create_window and delete_window procedures can be defined that make a series of system calls to an OS screen service in order to create or delete a window.

An advantage of this approach is its simplicity. A disadvantage is that as procedure calls are independent, they cannot be history-sensitive. For example, the create_window procedure is not aware of how many times it has been called in the past and hence how many windows have already been created. A related disadvantage is that operations on the same object (for example, 'resize', 'delete') cannot be synchronized.

4.8.2 Messages

Operations on ADTs can be defined as messages accepted by a long-lived *filter* process (Section 4.3). The long-lived process itself communicates with OS services. For example, a 'window' process may possess a circuit to a screen service and accept messages 'create', 'delete', etc.

An advantage of this approach is that the filter process, being long-lived, can maintain a history of interactions. Its single input stream permits it to synchronize operations. A disadvantage is that the long-lived process may constitute a bottleneck, particularly on a multiprocessor. Also, every application process that wishes to access the ADT must possess a reference to the process' message stream.

4.8.3 Exceptions

Operations on ADTs can be defined as exception messages processed by a task supervisor. This is the approach used to implement system calls in the Parlog OS described in Section 4.7.

This approach combines features of both procedures and messages. A task supervisor can be history sensitive. Yet application programs do not need to possess references to message streams to the supervisor.

The three approaches to implementing abstractions are illustrated in Figure 4.12.

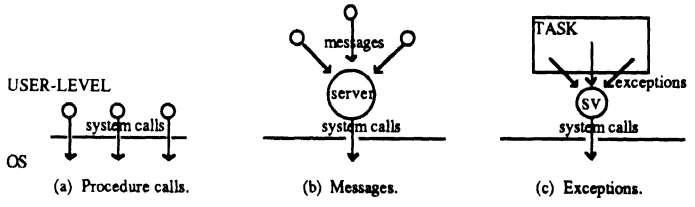


Figure 4.13 Approaches to implementing abstractions.

4.9 Summary

The essential components of the OS design methodology presented in this chapter can be summarized as follows:

- Operating systems are assumed to consist of a *kernel* that provides mechanisms used to implement the *operating system*, which in turn supports *user-level* programs.
- Operating systems are implemented as networks of concurrent processes that communicate and synchronize by means of shared logical variables.
- OS interfaces to kernel mechanisms are implemented in terms of language features: procedure call (primitives), unification (streams) and control metacall (status and control messages).
- Special processes termed *services* encapsulate both physical and logical resources. Processes that wish to access resources send messages to services. This permits accesses to be serialized when required.
- Application programs are executed as separate tasks using Parlog's control metacall. This protects the OS from application program failure and provides a means of monitoring and controlling their execution.

- An extension to Parlog, the control metacall *exception* message, is used to notify the OS of exceptional conditions in application programs. A *continuation variable* associated with an exception permits the OS to indicate how processing is to continue following the exception.
- System calls are implemented using the exception message. An OS process termed a *supervisor* is associated with a task. This translates exceptions representing system calls into messages to services. The continuation variable provides for error reporting.
- System calls support both remote procedure call and circuit interfaces to OS services.
- Naming schemes are implemented in the language by name servers that route streams of tagged messages to their destination.
- Simple validation techniques that control the sharing of variables between application and OS programs provide robust system services.
- User-level programs can construct more powerful abstractions using OS services in three ways: using procedures, messages and exceptions.

A simple Parlog OS that integrates these components has been presented. The design and implementation of a Parlog OS kernel is considered in the next chapter. Chapter 6 discusses the extension of these techniques to multiprocessors.

CHAPTER 5.

A Kernel Language

Parlog's high-level features permit a systems programmer to ignore implementation details. Instead, he can concentrate on specifying operating systems at a more abstract level. However, if Parlog operating systems are to be used to control computers, all language features used to implement them must be converted into features of a Parlog implementation. This requires the assistance of a kernel.

A Parlog operating system kernel, as illustrated in Figure 4.1, acts as an intermediary between a Parlog operating system and an underlying machine. It serves two principal purposes. First, it provides run-time support for Parlog language features that cannot be compiled directly to machine instructions. It thus implements a 'Parlog machine', capable of executing Parlog programs. Second, it performs machine-dependent functions that cannot conveniently be programmed in Parlog. For example, interrupt handling and low-level control of devices. A Parlog kernel may be implemented in machine code, microcode or hardware, and is replicated at each node in a multiprocessor.

In Section 4.1, it was observed that the design of an OS kernel must trade off functionality, efficiency, flexibility and simplicity. The nature of these tradeoffs clearly depends on both the OS to be supported and the underlying hardware. This chapter presents an approach to kernel design and implementation that emphasizes simplicity and flexibility while seeking to permit efficient implementation. The approach is targeted to multiprocessors (Section 1.2.1). It provides the functionality required by the OS outlined in Chapter 4. The presentation is otherwise OS and architecture independent.

Section 5.1 reviews the kernel functionality required by the Parlog OS outlined in Chapter 4. Sources of complexity are identified and a layered structure is proposed for kernel implementation. This provides direct support for a simple **kernel language**. The kernel language consists of the and-parallel subset of Parlog — Flat Parlog — plus simple metacontrol primitives. More complex kernel functions, and Parlog itself, are implemented in this kernel language.

Section 5.2 describes the kernel language and presents experimental results that motivate its selection. Section 5.3 outlines an approach to its uniprocessor

implementation. Section 5.4 describes the design, application and implementation of the kernel language's metacontrol primitives. Section 5.5 describes the kernel support required for multiprocessor execution of the kernel language. Distributed unification algorithms for Parlog are presented. The final section compares and contrasts this approach to kernel design with other, related work.

5.1 A Parlog Kernel

The Parlog OS outlined in Chapter 4 requires support from a kernel for the following functions:

Parlog: The execution of the basic language, including the or-parallelism (Section 2.5.6) that arises when user-defined procedures are called in guards.

Metacontrol: The creation, monitoring and control of tasks.

Processor scheduling: The allocation of processor resources to tasks, as influenced by the priority(_) metacall control message (Section 4.2.4).

Interrupt handling and device control: The signalling of run-time errors as exceptions. The translation of device input and other events into messages on streams. The implementation of low-level device drivers which may be invoked using Parlog primitives.

Memory management: Including mechanisms for signalling how much free memory is available and for aborting tasks when little free memory is available (Section 4.2.5).

If Parlog is to be used to program multiprocessor operating systems, the following functions must also be supported:

Distributed Parlog: The execution of Parlog on multiprocessors.

Distributed metacontrol: The creation, monitoring and control of tasks distributed over several nodes on a multiprocessor.

Multiprocessor scheduling: The allocation of processor resources to tasks distributed over several nodes on a multiprocessor.

This set of functions includes some that are simple and others that are relatively complex. A kernel that implements them all is likely to be both complex and inflexible.

It thus appears desirable to find some subset of these functions that can be used to implement the rest.

The interrupt handling and device control functions of a Parlog kernel can be programmed using conventional techniques [Peterson and Silberschatz, 1983; Joseph *et al.*, 1984]. The only novel feature of a Parlog kernel's implementation of these functions is the interface it must provide to Parlog language features. These functions are hence not dealt with here.

Memory management is a more complex problem. However, as it is closely linked with problems of language implementation, it is not treated explicitly herein. (But see Section 5.6).

This leaves three significant areas of complexity: Parlog itself, metacontrol and processor scheduling.

Parlog itself is complex, despite the apparent simplicity of the process pool computational model described in Section 3.2.2. This model only directly supports the and-parallel subset of Parlog. Full Parlog also incorporates or-parallelism and other language features (sequential operators and negation) that require support for a process hierarchy and hence complex run-time data structures and control algorithms in an implementation [Gregory *et al.*, 1988].

Metacontrol functions such as abortion, suspension, termination detection and deadlock detection become complex when tasks may be nested or distributed over many nodes in a multiprocessor. The control metacall, as defined in Chapters 3 and 4, is also inflexible: for example, its status stream centralizes exception handling, which in a distributed task might be more efficiently handled at individual nodes.

Processor scheduling is perhaps the most complex function because the least well defined. It includes the problems of scheduling tasks on individual nodes, allocating nodes to tasks and mapping processes to nodes.

To avoid encoding this complexity in the kernel, it is proposed that the kernel support a simpler **kernel language**. More complex functions are programmed in this kernel language. This approach leads to a simpler, more reliable kernel. It can also lead to a more flexible language, as the kernel can provide a number of simpler mechanisms rather than inflexible, monolithic primitives such as the control metacall. If the kernel language is well chosen, it need not compromise efficiency. However, it cannot be *too* simple: it must support all functions required by OS programs.

This leads to a layered organization for the kernel and operating system, illustrated in Figure 5.1.

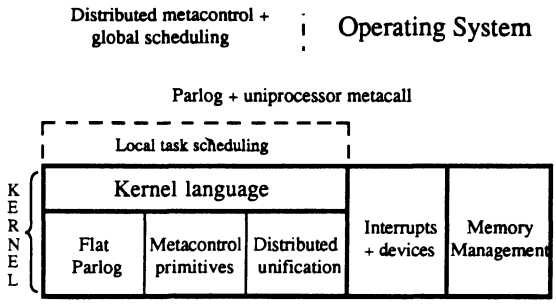


Figure 5.1 Kernel and operating system organization.

The kernel language consists of the and-parallel subset of Parlog, *Flat Parlog*, augmented with simple *metacontrol primitives*. Kernel support for *distributed unification* permits Parlog processes located on different nodes in a multiprocessor to communicate by unifying shared variables. The kernel also provides *interrupt handling*, *device drivers* and *memory management*; as noted above, these issues are not dealt with here.

The kernel language's metacontrol primitives provide support for the creation, monitoring and control of uniprocessor tasks: tasks executing on a single node. An interesting feature of the metacontrol primitives presented here is that they permit *local* (that is, *uniprocessor*) *task scheduling* algorithms to be programmed in the kernel language.

Parlog itself plus a *uniprocessor control metacall* can be compiled directly to the kernel language. *Distributed metacontrol* and *global* (that is, *multiprocessor*) *scheduling* can be programmed in the kernel language. The implementation of these functions is discussed in Chapter 6.

5.2 The Kernel Language

5.2.1 Flat Parlog

In Flat Parlog, clause guards can only contain calls to primitive, non-recursive procedures implemented in the language kernel. There is hence no need to maintain a process hierarchy; the state of a computation is a flat conjunction or *pool* of processes. Clause selection in Flat Parlog can be performed efficiently by a sequential algorithm: there is hence no need for parallel clause selection in an implementation.

Here, Flat Parlog is further restricted, following Gregory [1987]. The *guard* of a Flat Parlog clause can only contain **guard primitives**. Guard primitives are kernel primitives that may suspend but that cannot side-effect their environment. For example: data, =.. and var. The *body* of a Flat Parlog clause consists of a (possibly empty) *sequential* conjunction of **body primitives** followed sequentially by a (possibly empty) *parallel* conjunction of calls to user-defined procedures. Body primitives cannot suspend but may side-effect their environment. For example: write, read and =. A Flat Parlog clause thus has the form:

$$P \leftarrow G_1, G_2, ..., G_l \cdot B_1 \,\&\, B_2 \,\&\, ... \,\&\, B_m \,\&\, (C_1, C_2, ..., C_n). \qquad k, m, n \geq 0$$

where the G_i are guard primitives, the B_j are body primitives and the C_k are calls to user-defined procedures.

These restrictions on the form of a Flat Parlog program permit the use of an evaluation strategy based on the process pool computational model described in Section 3.2.2. Recall that in this model, a reduction attempt evaluates the guards of clauses in a procedure in an attempt to find a clause capable of reducing a process. If all guards suspend, the process is placed back in the process pool. As guards cannot contain side-effecting primitives or calls to user-defined procedures, a subsequent reduction attempt involving the same process can safely and efficiently reevaluate clauses; there is no need to save the state of a suspended evaluation. This simplifies implementation.

Once a clause has been selected to reduce a process, the sequence of body instructions can immediately be executed. As these cannot suspend, there is again no need to save an intermediate state. Once the body instructions are executed, the parallel conjunction of body goals can be added directly to the process pool.

Flat Parlog's sequential conjunction of body primitives permits the sequencing of side-effects; as noted in Section 3.2.4, this is the primary application of Parlog's sequential conjunction operator. Flat Parlog does not provide direct support for Parlog's negation operator or or-parallelism. However, Parlog programs that use these features can be translated to Flat Parlog programs using simple transformation techniques [Gregory, 1987]. Transformed programs represent the process hierarchies implied by these language features as sets of and-parallel processes. These processes synchronize and communicate by means of additional status and control variables.

5.2.2 Metacontrol Primitives

The kernel language's metacontrol primitives support the creation, monitoring and control of *uniprocessor* tasks: tasks that execute on a single node in a multiprocessor. Each primitive encodes some subset of the functionality represented by Parlog's control metacalls.

Recall that the control metacalls described in Section 4.2.6 have the general form:

```
call(Goal, Status, Control)
call(Module, Goal, Status, Control)
```

They support control messages such as stop and priority(_).

These metacalls (and their control messages) in fact describe four largely orthogonal control functions:

- the creation of a task: an entity whose execution can be monitored and controlled using Status and Control variables.
- subsequent control of this task (suspend, resume, abort).
- access to the dictionary contained in Module to find the code associated with the relation named by Goal; execution of this code.
- the scheduling of a task with a specified priority.

These functions can be encoded using simpler primitives:

'TASK'(S↑, TR↑): creates a uniprocessor task with status stream S. This task has a single process, which continues to execute the current clause. (The original task — the task in which the TASK primitive was executed — continues to execute other processes). The TASK primitive generates a **task record**, TR. This is a language term representing the new task, that can be used to refer to it subsequently. A call to this primitive *succeeds immediately*; once created, a task executes independently of the task that created it. Status messages include succeeded, stopped, exception/3, deadlock/1 and undeadlock. Unification and process failure are signalled as exceptions; they do not cause failure of the task in which they occur.

'SUSPEND'(TR?), 'CONTINUE'(TR?), 'STOP'(TR?): which suspend, resume and stop the task represented by the task record TR respectively.

Module?@Goal?: accesses the dictionary associated with Module to find the code associated with Goal and begins executing that code.

'CALL'(Goal?): accesses the dictionary associated with the current module (the module in which the procedure that calls this primitive is located) to find the code associated with Goal and begins executing that code.

'PRIORITY'(TR?, P?): associates a priority P with the task represented by the task record TR.

'RUN'(TR?, TSlice?): instructs the kernel to execute the task with task record TR for a period TSlice.

The mode annotations indicate whether an argument must be supplied at the time of call (?) or is generated by the primitive (↑).

The following observations can be made about these primitives:

- They do not support nested metacalls. Monitoring and control of nested tasks must be programmed in the kernel language
- As process and unification failure are signalled as exceptions rather than as task failure, failure is not catastrophic. Exception handlers and debuggers that deal with failure can be programmed in the kernel language.
- The significance of the priority associated with a task by the PRIORITY primitive is not specified here; as will be described in Section 5.4, this is interpreted by a task scheduler written in the kernel language. The use of the RUN primitive is also described in that section.

5.2.2.1 Implementing Control Metacalls

An attractive feature of these metacontrol primitives is that they can be used to implement a range of metacontrol functions.

Program 5.1 implements the three and four argument metacalls described in Section 4.2.6. call(M,G,S,C) initiates controlled execution of a goal G using a module M. call(G,S,C) initiates controlled execution of a goal G defined in the same module as the procedure which makes the call.

call/4 calls a procedure run/4 which uses the TASK primitive to become a controllable task with task record TR and status stream S (C1,2). The new task then proceeds to call the @ primitive, which accesses the module M and starts to execute the goal G (C2). Note the sequential conjunction operator conjoining the TASK and @ calls; this sequential control flow is essential to the correct execution of this program. The task record and status stream of the new task are passed to a concurrently executing monitor process (C1). This monitors the control variable supplied in the original

metacall and translates messages stop, suspend, continue and priority(_) into calls to the STOP, SUSPEND, CONTINUE and PRIORITY primitives with the task record as an argument (C5-8). stop, suspend and continue are echoed on the metacall's status stream.

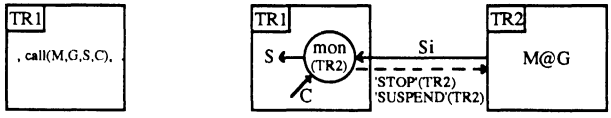
```
mode call(Module?, Goal?, Status↑, Control?), call(Goal?, Status↑, Control?),
     run(Module?, Goal?, Status↑, TaskRecord↑), run(Goal?, Status↑, TaskRecord↑),
     monitor(StatusIn?, StatusOut↑, Control?, TaskRecord?).

call(Module, Goal, S, C) ← run(Module, Goal, Si, TR), monitor(Si, S, C, TR).          (C1)

run(Module, Goal, S, TR) ← 'TASK'(S, TR) & Module@Goal.  % Create new task.          (C2)

call(Goal, S, C) ← run(Goal, Si, TR), monitor(Si, S, C, TR). % 3-argument call.      (C3)

run(Goal, S, TR) ← 'TASK'(S, TR) & 'CALL'(Goal).          % Create new task.         (C4)

monitor(Si, stopped, stop, TR) ← 'STOP'(TR).              % Abort execution.         (C5)
monitor(Si, [suspend |S], [suspend |C], TR) ←             % Suspend execution.       (C6)
     'SUSPEND'(TR) & monitor(Si, S, C, TR).
monitor(Si, [continue |S], [continue |C], TR) ←           % Resume execution.        (C7)
     'CONTINUE'(TR) & monitor(S, S, C, TR).
monitor(Si, S, [priority(P) |C], TR) ← 'PRIORITY'(TR, P) & monitor(Si, S, C, TR).   (C8)
monitor(succeeded, succeeded, C, TR).                     % Termination.             (C9)
monitor([exception(failure, _, _) |Si], failed, C, TR) ← 'STOP'(TR).                (C10)
monitor([M |Si], [M |S], C, TR) ←                         % Forward other status.    (C11)
     M =/= exception(failure, _, _) : monitor(Si, S, C, TR).
```

Program 5.1 Three and four argument metacalls.

Recall that the status messages stopped, suspend and continue should be echoed only *after* the control function represented by the corresponding control message has taken effect (Section 3.4). These status messages can be echoed immediately here because the metacontrol primitives STOP, SUSPEND and CONTINUE are defined to take immediate effect on the uniprocessor task they are applied to. These messages *cannot* be echoed immediately when a task is distributed across several nodes on a multiprocessor (see Section 6.1).

monitor also processes status messages received from the new task. Termination of the task — signalled by the status succeeded — causes the monitor to terminate (C9).

Recall that process and unification failure are signalled in the kernel language as exceptions (with type failure). The metacall's failed status message is implemented by monitoring the task's status stream and, when such exceptions are detected: (a) generating a failed message and (b) aborting the task (C10). Other status messages are passed out on the task's status stream (C11).



(a) Before (b) After

Figure 5.2 Implementation of the control metacall.

Figure 5.2 illustrates the implementation of the control metacall. A single task exists before a call to call/4. After the call, two tasks exist. The 'parent' task (TR1) contains a monitor process which possesses references to the task record representing the second task, TR2, and to its status stream, Si. It monitors both this status stream, Si, and the control stream, C, and generates a filtered status stream S. It translates control messages into calls to metacontrol primitives.

A range of control metacalls providing more or less functionality can be programmed in this way. For example, a metacall fcall/4 can be defined that is equivalent to call/4 but uses a monitor procedure that does not contain clause C10. Process and unification failure are then signalled as exceptions. An application of this metacall is illustrated in Section 6.1.

A more complex monitor procedure can implement control of nested metacalls. The monitor must trap metacalls made by 'offspring' tasks as exceptions. The monitor then creates a new task, plus a new monitor which responds to control messages directed either to it or to its parent. The tree of nested metacalls is thus simulated using a flat pool of tasks and a tree of monitor processes. The executable specification for the control metacall given in Appendix I illustrates how this is achieved.

5.2.2.2 Implementing a Control Call

Program 4.5 includes the clause:

```
try(B1, B, Cont, G) ← call(copy(B1,B), S, C), result(So, Cont, G).
```

call/3 is not used here as a metacall, as its first argument is instantiated. Gregory

[1987] terms this use of call/3 a **control call**.

No dictionary lookup is required to implement a control call. This example can hence be compiled to:

```
try(B1, B, Cont, G) ←
        copy(B1, B, S, TR), monitor(S, So, _, TR), result(S, Cont, G).

copy(B1,B, S, TR) ← 'TASK'(S,TR) & copy(B1,B).
```

The procedure copy/4 uses the TASK primitive to become a new task and proceeds immediately to execute the procedure copy/2. The dictionary lookup implicit in the three argument metacall is avoided.

Note that as the control call is used here only to monitor — not to control — the call to copy/2 , the full functionality of monitor/4 (defined in Program 5.1) is not required: only clauses C9-11 are needed. Also, the functionality of these clauses and of result/3 (Program 4.5) can be combined to give a single procedure.

5.2.3 Why Flat Parlog

The ability to call user-defined procedures in guards (**deep guards**) provides Parlog with a limited form of or-parallel search. This or-parallelism can be implemented, as noted in Section 5.2.1, by compilation to the and-parallel subset of Parlog. Alternatively, it may be supported directly in the language. The latter approach provides potentially more efficient support for or-parallel evaluation but complicates the implementation.

To determine whether it is useful for a Parlog OS kernel language to provide direct support for or-parallel evaluation, tests were performed to determine the extent to which Parlog system programs make use of this language feature. These tests consisted of static and dynamic analyses of three Parlog system programs written by different programmers. The programs analysed are a Parlog compiler (*Compiler*), a real time process control program (*Train*: a model train set controller) and a planning program with a graphics component (*Taxi*: a taxi scheduler). They are substantial applications which are believed to be representative of Parlog system programs.

Parlog procedures may be classified as flat, if-then-else and search. The guards of the clauses defining a **flat** procedure may not contain any calls to user-defined procedures. Calls to flat procedures do not therefore result in significant or-parallel evaluation. In **if-then-else** procedures, a user-defined procedure is used in a clause

guard to determine whether that clause or a subsequent default clause should be used to reduce a process. The default clause — separated from the clause with the deep guard by a sequential clause search operator — is selected if the test fails. Deep guards are used in if-then-else procedures to make programs more succinct and readable, rather than to perform or-parallel search. Finally, in **search** procedures, two or more clauses not separated by sequential clause search operators contain deep guards; true or-parallel evaluation can thus arise.

Program 5.2 gives examples of flat, if-then-else and search procedures. on_list(L,E) has the logical reading: E is a member of list L. on_tree(T,K,V) reads: the tuple {K,V} is a member of a tree T. service(Rs,E) reads: Rs is a list of pairs {L,R}, where for each pair, *either* L is a list containing the element E and R = true *or* L is not such a list and R = false. The if-then-else guard in service uses on_list to determine whether the first or second clause is used to reduce a process.

Flat

```
mode on_list(List?, Element?).

on_list([H |T], E) ← H =/= E · on_list(T, E)
on_list([E |T], E)
```

Search

```
mode on_tree(Tree?, Key?, Value↑).

on_tree(t(L,R),K,V) ← on_tree(L,K,V) · true.
on_tree(t(L,R),K,V) ← on_tree(R,K,V) · true.
on_tree(leaf(K,V),K,V).
```

If-then-else

```
mode service(Requests?, Element?).

service([{L,R} |Rs], E) ←
    on_list(L, E) :
    R = true, service(Rs, E);
service([{L,C} |Rs], E) ←
    R = false, service(Rs, E).
```

Flattened if-then-else

```
mode flat_service(Requests?, Element?),
    on_list(List?, Element?, Result↑).

flat_service([{L,R} |Rs], E) ←
    on_list(L, E, R), flat_service(Rs, E).

on_list([H |T], E, R) ← H =/= E : on_list(T, E, R).
on_list([E |T], E, true).
on_list([ ], _, false).
```

Program 5.2 Types of Parlog procedures.

Program 5.2 also defines a procedure flat_service that performs the same computation as service, using flat guards. on_list(L,E) is redefined as on_list(L,E,R), which reads: *either* list L contains E and R = true, *or* list L does not contain E and R = false. It is suggested that flat_service is more clumsy that service and less obviously

correct; this illustrates the utility of the if-then-else construct.

Note that service can be rewritten as a search procedure. The sequential clause search operator is changed to the usual parallel operator and the negated call not(on_list(L,E)) is added to the guard of the second clause. The resulting program is however less efficient, as the call on_list(L,E) is evaluated twice, to no purpose. Procedures in the programs tested that used negation in this way, when an if-then-else structure could have been used instead, were thus transformed to use sequential clause search operators.

Figure 5.3 represents the process hierarchies created by calls to these procedures. on_list/2 creates a single process, which checks each member of a list in turn. service/2 creates a process which creates a single on_list/2 or-process. on_tree/3 creates a tree of or-parallel on_tree/3 processes to check all subtrees of a tree in parallel. It is maintaining the latter type of hierarchy that complicates an implementation.

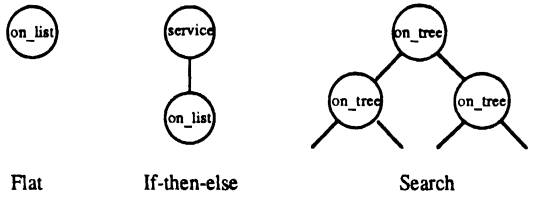


Figure 5.3 Types of process hierarchy.

Table 5.1 presents the results of a static analysis of the three system programs and, for purposes of comparison, Program 5.2. For each program, the number of flat, if-then-else and search procedures is given, as is the total number of procedures in the program. The number of distinct procedures that are called in guards is also given, under the heading *Called*.

	Flat	If-then-else	Search	Total	Called
Compiler	184	47	0	231	14
Train	46	4	1	51	4
Taxi	103	7	1	111	12
Program 5.2	3	1	1	5	2

Table 5.1 Static analysis of Parlog programs. (No. of procedures).

Tables 5.2 and 5.3 presents the results of dynamic analyses of the same programs. These results were obtained by instrumenting a Parlog implementation and executing the programs on typical input data.

Table 5.2 gives the number of calls to user-defined procedures in guards (*Guard Calls*) and the number of reductions that occurred as a result of these calls (*Guard Reductions*). For example, a call service([{[2,3,1],R}], 1) results in 1 guard call and 3 guard reductions: on_list/2 is called once, to process the request {[2,3,1],R}, and performs 3 reductions. Nested calls to deep guards are counted as a single guard call. Thus a call on_tree(T,K,V), T a deeply nested tree, generally results in two guard calls (one per 'deep' clause) and some larger number of guard reductions.

	Total Calls	Guard Calls	Guard Reductions	Calls/ Total (%)	Reductions/ Total (%)
Compiler	29.7	1.15	15.1	4	51
Train	22.4	4.13	4.3	18	19
Taxi	14.4	0.32	5.9	2	41

Table 5.2 Dynamic analysis of Parlog programs. (1000's of calls).

Table 5.3 characterizes guard calls by giving the mean, median, maximum and standard deviation for the number of reductions per guard call.

	Mean	Median	Maximum	σ
Compiler	13.1	13	244	20.9
PASS	1.03	1	5	0.30
Taxi	18.8	3	362	56.0

Table 5.3 Dynamic analysis of Parlog programs: guard call distribution.

Two conclusions may be drawn from these results. First, deep guards, whilst widely used, are rarely used to perform significant computation in systems programs. 2-18% of all calls invoke procedures in guards; 19-51% of all reductions occur as a result of these calls. Yet the mean number of reductions per guard call is low (1-19) and the medians even lower (1-13). The high standard deviations in *Compiler* and *Taxi* indicate a highly skewed distribution: in *Taxi*, the distribution is skewed by a single procedure that applies an iterative algorithm to compute a square root.

The second conclusion is that system programs use or-parallelism very rarely. Only two search procedures were found in the programs tested. Dynamic analysis did not provide information on the amount of or-parallel evaluation performed. However, inspection of the two search procedures shows that in each case or-parallel guard

evaluation amounts to only one or two reductions.

These results suggest that and-parallelism is the dominant programming paradigm in Parlog system programs. Deep guards are used primarily as a useful shorthand for 'if-then-else' structures and rarely, if at all, to perform or-parallel evaluation. When used as shorthand for if-then-else structures, they generally perform a simple, computationally inexpensive test.

Applications to be supported by a Parlog OS may of course require or-parallel evaluation. However, it is believed that Parlog's or-parallelism is unlikely to be important even there, due to its inherent inefficiency. Though Parlog's deep guards permit or-parallel evaluation of alternative clauses, committed-choice evaluation means that only a single solution can be found for a problem. For example, a call on_tree(T,K,V) will only find a single value for V even if T contains several entries {K,V1}, {K,V2}, etc. Work spent searching for other solutions is hence wasted, as this work is abandoned as soon the first solution is found. If or-parallel evaluation is required in Parlog, it is probably preferable to either compile it to and-parallel evaluation or to provide a set-constructor interface to an all-solutions or-parallel language (Section 3.5.3).

It is therefore argued that a purely and-parallel language is adequate as a kernel language for a Parlog OS. As non-flat if-then-else guards are widely used in Parlog system programs, they should be supported efficiently. However, this does not require support for deep guards in an implementation: as noted above, it can be achieved by transformation to Flat Parlog. As the number of procedures called in guards is low (6-11% of all procedures in the programs analysed above) the overhead of transforming them should not be excessive.

5.2.4 Why Metacontrol Primitives

It has been proposed that the metacontrol functions represented by the kernel language's metacontrol primitives be supported by the kernel of a Parlog OS. An alternative approach to metacontrol implements these functions by program transformation. This approach has the advantage of not complicating language semantics. However, benchmark results presented here suggest that program transformation is significantly more expensive than equivalent kernel mechanisms. As efficient support for a range of metacontrol functions is vital to a Parlog OS, it is argued that these functions are best implemented in the kernel.

5.2.4.1 Metacontrol through Transformation

Hirsch *et al.* [1986] propose that metacontrol functions be implemented by program transformations that extend programs with additional code that reports termination, permits control, etc. This approach is illustrated using a simple example.

Program 5.3 implements two versions of a program reverse. The logical reading of reverse(Xs,Ys) is: Ys is the list Xs reversed. The logical reading of reverse(Xs,Ys,S) is: Xs is a list and Ys is Xs reversed, and S = succeeded, *or* Xs is not a list and S = failed. Operationally, a call reverse(Xs,Ys,S) performs the same computation as a call reverse(Xs,Ys), but in addition (a) reports successful termination by binding the status variable S to succeeded (C2,5) and (b) reports failure by binding S to failed (C3,6). smerge processes combine the statuses of conjunctions to produce a final status.

More comprehensive transformations that permit evaluation of a procedure to be controlled are also defined by Hirsch *et al.*

```
mode  reverse(Xs?,Ys↑),
      append(Xs?,Ys?,Zs↑).

reverse([X | Xs], Ys) ←
    reverse(Xs, Zs),
    append(Zs, [X], Ys).
reverse([ ], [ ]).

append([X | Xs], Ys, [X | Zs] ) ←
    append(Xs, Ys, Zs).
append([ ], Ys, Ys).
```

Original

```
mode   append(Xs?,Ys?,Zs↑,S↑),
       reverse(Xs?,Ys↑,S↑), smerge(S1?,S2?,S↑).

reverse([X | Xs], Ys, S) ←                          (C1)
    reverse(Xs, Zs, S1),
    append(Zs, [X], Ys, S2), smerge(S1, S2, S).
reverse([ ], [ ], succeeded);      % Note ';'.     (C2)
reverse(Xs, Ys, failed).                            (C3)

append([X | Xs], Ys, [X | Zs] , S) ←                (C4)
    append(Xs, Ys, Zs, S).
append([ ], Ys, Ys, succeeded); % Note ';'.         (C5)
append(Xs, Ys, Zs, failed).                         (C6)

smerge(succeeded, succeeded, succeeded).(C7)
smerge(failed, _, failed).                          (C8)
smerge(_, failed, failed).                          (C9)
```

Transformed

Program 5.3 Termination detection through transformation.

5.2.4.2 Comparison: Performance

Appendix II describes an experimental study that compares the performance of metacontrol functions when implemented both using kernel mechanisms and through transformation. The results of this study are summarized here.

Any metacontrol mechanism necessarily introduces some run-time overhead. This may take the form of increased code size, CPU time requirements and run-time memory requirements. To quantify overheads associated with program transformation, transformations defined by Hirsch *et al.* [1986] were applied to a number of benchmark programs. These produced programs that reported process failure and successful termination in a task and that could be suspended, resumed or aborted. The performance of the transformed and untransformed programs when executed on a uniprocessor Flat Parlog implementation was determined. This implementation is an emulator for an abstract machine (described in Section 5.3), written in the C programming language. The size of the transformed programs was also measured.

The Flat Parlog implementation was then extended to support the metacontrol functions implemented by transformation. The performance of the untransformed programs when executed on the modified implementation was measured.

Additional run-time memory requirements due to both kernel support and transformation were also measured, but were not found to be a significant source of overhead.

In summary, referring to the most substantial benchmark program, the compiler:

- the performance degradation associated with kernel support is 2 %
- the performance degradation associated with transformation is 15 %
- kernel support does not effect code size
- transformation increases code size 98 %
- run-time memory requirements are not a significant source of overhead

This study indicates that given existing kernel implementation and Parlog compilation techniques, kernel support incurs substantially less overhead than program transformation. The efficiency of both approaches can of course be improved, by microcoding kernel functions or improving compilation techniques, for example.

5.2.4.3 Comparison: Expressiveness

The extended Parlog implementation for which benchmark results are summarized above also implements deadlock detection and round-robin task scheduling. Program transformations that provide efficient implementations of these functions have not been described. Kernel support thus appears to be in absolute terms *more expressive* than program transformation. This should not be surprising. Modifications to a language kernel can be used to define arbitrary extensions to a language's operational semantics. They can certainly define extensions that would be hard to define efficiently in the original language. A Parlog kernel, for example, can maintain a scheduling structure that permits it to reduce processes belonging to a particular task, or to determine whether a task has reducible processes (Section 5.4). This permits inexpensive implementations of task scheduling and deadlock detection mechanisms. Without access to such centralized structures, these functions are complex and expensive (see Section 6.2).

5.3 Uniprocessor Implementation of Flat Parlog

In order to make this chapter self-contained, the approach to Flat Parlog implementation described in [Foster and Taylor, 1988] is described briefly here. This is based on a uniprocessor abstract machine for Parlog named the Flat Parlog Machine (FPM). An abstract machine is a machine architecture in which low-level implementation details are not completely specified. It is thus possible to implement it in a number of different ways.

The FPM implements the process pool computational model described in Section 3.2.2. Recall that in this model, the state of a Parlog computation is represented as a pool of processes. Evaluation proceeds by repeatedly selecting processes from this pool and attempting to reduce them. The FPM represents the state of a computation in terms of this model and executes instructions that encode basic operations on this computation state. Programs to be executed on the FPM are encoded as sequences of FPM instructions.

Subsequent sections describe extensions to the FPM which implement the kernel language's metacontrol primitives (Section 5.4) and support parallel execution (Section 5.5).

5.3.1 Computational Model

The FPM implements the process pool computational model described in Section 3.2.2, with two important optimizations. It introduces a **scheduling structure** to avoid the overhead of repeatedly attempting to reduce suspended processes (busy waiting) and supports **tail recursion**, to avoid the overhead of selecting a process from the process pool at every reduction.

The scheduling structure consists of a single **active queue** containing reducible processes plus multiple **suspension lists** which link together processes that require particular data. Suspension lists enable suspended processes to be located and moved to the active queue when the variable that they are suspended on is instantiated. The suspension structure used permits a process to be suspended on more than one variable.

The tail recursion optimization is a generalization of that commonly used in Prolog implementations. When a clause with body goals is used to reduce a process, reduction can continue with one of those goals. This saves the overhead of adding that process to the process pool and subsequently selecting it.

Tail recursion optimization is only applied a finite number of times before the current process is moved to the end of the active queue and a new process selected for reduction. The number of tail recursive calls permitted before such a **process switch** occurs is a **process timeslice**. This provides *And-justice* (Section 3.2.6): that is, it guarantees that any process capable of being reduced will eventually be reduced.

When trying to reduce a process, the FPM tries each clause in the associated procedure in turn. Each such **clause try** may suspend, succeed or fail. If any clause try succeeds, that clause is used to reduce the process. If a clause try suspends, because data is not available, the variable(s) for which it requires values are recorded. If no clause try succeeds, but at least one clause try suspends, the process is linked into the suspension lists of all recorded variables. If all clause tries fail, the process try and hence the Flat Parlog computation fails.

5.3.2 Machine Architecture

The principal data area in the FPM is a **heap**. This holds tagged words representing both Parlog data structures (terms) and **process records** representing Flat Parlog processes. Valid data types are variables, constants (integers, strings and the special constant nil, which represents the empty list), structured terms (tuples and lists),

variables and references to other data structures. Process records contain pointers to the code that the process is executing, its sibling in the scheduling structure and a fixed number of arguments.

The only other data structure used by the Flat Parlog machine is the **suspension table** used during a reduction attempt to record variables to suspend upon if no clause try succeeds.

The current state of the abstract machine is recorded in various registers. These form three distinct groups according to the time at which their values are relevant. General registers are used for storing global aspects of the machine state. Process registers are only used during a reduction attempt. Clause registers are used at each clause try. Registers relevant to the present discussion are:

General Registers

- QF Queue Front, points to the first process in the active queue.
- QB Queue Back, points to the last process in the active queue.

Process Registers

- CP Current Process, points to the process currently being reduced.
- TS Time Slice, the remaining time slice for the current process.
- PC Program Counter, the instruction pointer. Contains the address of the next instruction to be executed.

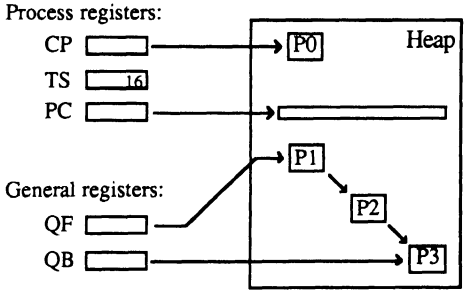


Figure 5.4 Flat Parlog Machine (FPM) architecture.

Figure 5.4 represents the data structures of the FPM. In this figure, the active queue is depicted as containing three process records, labelled P1, P2 and P3. A

process P0 is currently being reduced. This has a remaining timeslice of 16.

5.3.3 Abstract Instruction Set

The instruction set of the Flat Parlog machine comprises **unification** instructions that implement primitive unification operations generated when programs are compiled to standard form (Section 3.2.3) and **control** instructions, used to encode Flat Parlog's computational model.

A complete specification of these instructions can be found in [Foster and Taylor, 1988].

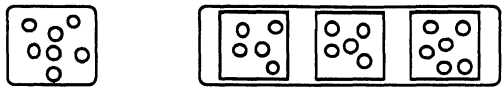
5.4 Uniprocessor Implementation of Metacontrol

An attractive feature of the kernel language's metacontrol primitives (Section 5.2.2) is that their implementation in a language kernel need not be overly complex. Modifications to the Flat Parlog Machine (Section 5.3) required to support them are described here. Recall that these primitives support the creation, monitoring and control of a task executing *on a single node* in a multiprocessor.

5.4.1 Extensions to Computational Model

Introducing the notion of task into a purely and-parallel language such as Flat Parlog requires an extension to the process pool computational model described in Section 3.2.2. The set of processes that comprise a Parlog task need to be monitored and controlled as if they were a single entity. This suggests that each task should be viewed as a separate process pool. The computational model must thus allow for multiple process pools.

If nested metacalls were to be supported directly in the computational model, the state of a computation would be a recursively defined pool of tasks and processes. To avoid the need for such a complex structure, tasks are instead treated as independent entities. The state of a computation is represented as a pool of tasks, each of which is a pool of processes. The monitoring and control of nested tasks can be programmed in Parlog, if required.



(a) Process pool. (b) Multiple process pools.

Figure 5.5 Simple and extended computational models.

Figure 5.5 contrasts the process pool and extended computational models. Computation in the extended model proceeds by repeatedly selecting both a task and a process within that task, and attempting to reduce that process. A reduction attempt may succeed, suspend or fail as before. Failure is treated differently: instead of causing failure of the entire computation, an exception message (Section 4.2.2) is generated on the corresponding task's status stream and a continuation variable is placed in the task's process pool.

In this model, an empty process pool represents successful termination of a task. A non-empty process pool which contains no reducible processes represents deadlock.

The extended Flat Parlog Machine (eFPM) implements this extended computational model. It introduces optimizations similar to the FPM's tail recursion and scheduling structure (Section 5.3.1). These are the notions of **current task**, which avoids the overhead of selecting a task at each reduction, and **scheduler**, which avoids the overhead of repeatedly selecting tasks with no reducible processes for reduction.

A separate active queue is maintained for each task. The eFPM repeatedly reduces processes from the active queue of a current task nominated by the scheduler (see below) until a task timeslice (defined below) ends or certain events occur. A task switch then occurs and a new task becomes current.

5.4.2 Task Scheduling

Recall that in the FPM, a simple process scheduling structure and round robin scheduling algorithm are used to: (a) avoid the overhead of repeatedly selecting suspended processes for reduction and (b) ensure that reduction is And-just. The eFPM could incorporate a similar scheduling structure and algorithm for tasks. This would ensure: (a) that tasks with no reducible processes (deadlocked tasks) were not selected for reduction and (b) that no task was ignored indefinitely. (In fact, a round robin strategy can do more than this: it can ensure that processor resources are allocated fairly to all tasks).

Implementations of a round robin scheduling algorithm for the FPM, based on an active queue, and more complex scheduling structures, based on an active tree, are described in [Foster, 1987b]. Such fixed scheduling strategies have been used successfully in very simple systems (for example, Modula [Wirth, 1977]). In general, however, operating systems require more powerful and flexible scheduling algorithms. To avoid the complexity and inflexibility inherent in a kernel implementation of such algorithms, kernel mechanisms are provided that permit a range of scheduling algorithms to be programmed in Parlog.

The eFPM does not maintain any scheduling structure. Tasks are represented as data structures termed **task records**. The eFPM only retains a reference to the task record of the initial task created when the Parlog OS is initiated. This is termed the **scheduler**. It makes tasks created subsequently available to the scheduler by appending their task records to a **scheduler event stream** (SES). This stream is consumed by the scheduler. The scheduler, a Parlog program, is responsible for deciding which tasks should be executed and for how long.

The scheduler is initially the current task. It consumes the scheduler event steam (which includes the task records of any tasks it has created) and constructs a Parlog term containing these task records. This term represents a scheduling structure. It then selects a task from this structure and uses the metacontrol primitive RUN to requests the kernel to switch to executing this task. The kernel executes this task for the **task timeslice** specified in the RUN call (its second argument), or until the task terminates or deadlocks. It then appends the task record to the SES, if the task is still reducible, and switches to executing the scheduler. The kernel thus alternates between executing the scheduler and executing other tasks nominated by the scheduler. The kernel also appends task records to the SES when tasks that were deadlocked become undeadlocked due to data becoming available, when tasks that were suspended are resumed and when tasks have their priority changed.

An attractive feature of this approach to task scheduling is that it only requires simple kernel mechanisms. It is also flexible: a range of scheduling algorithms can be implemented as Parlog programs. These may be expected to be less efficient than equivalent algorithms implemented entirely in the kernel, as the scheduler must perform some small number of reductions each time a task switch occurs. This inefficiency is not important if a significant number of reductions is performed between task switches. However, it becomes expensive to use tasks to perform very simple operations.

Section 5.4.6 presents an example of a Parlog scheduler.

5.4.3 Extensions to Architecture

Recall that in the FPM, the current state of the abstract machine is recorded in three sets of registers: general registers, process registers and clause registers. The general registers include the queue head and queue tail registers, which point to the head and tail of the active queue.

In the eFPM, a separate active queue is maintained for each task. The head and tail of a task's active queue, and other information about the task, are maintained in a data structure much like a process record called a **task record**. Both the task record and its constituent fields are represented as Parlog terms. The fields of the task record include:

- QF Queue Front, points to the first process record in the task's active queue.
- QB Queue Back, points to the last process record in the task's active queue.
- PK Process Count, the number of processes in the task.
- MK Message Count, the number of outstanding messages. (See Section 5.5).
- SV Status Variable, points to the task's status variable.
- ST State, which indicates whether the task is active, suspended, aborted, etc.
- PR Priority, the priority associated with the task.

New machine registers termed **task registers** are used to buffer some of this data when a task is current. These include:

- CT Current Task.
- QF Queue Front, buffers the current task's QF field.
- QB Queue Back, buffers the current task's QB field.
- PK Process Count, buffers the current task's PK field.
- MK Message Count, buffers the current task's MK field.
- SL Task Slice, the remaining task timeslice for the current task.

The process record used to represent a process has an additional task (TA) field, which contains a reference to the process' task record.

The only general registers relevant to this discussion are:

- SCH SCHeduler, points to the task record of the scheduler task.
- SES Scheduler Event Stream, points to the uninstantiated tail of the scheduler event stream.

The only other new data structure is an interrupt vector that specifies kernel routines that handle run-time errors, timer interrupts, etc.

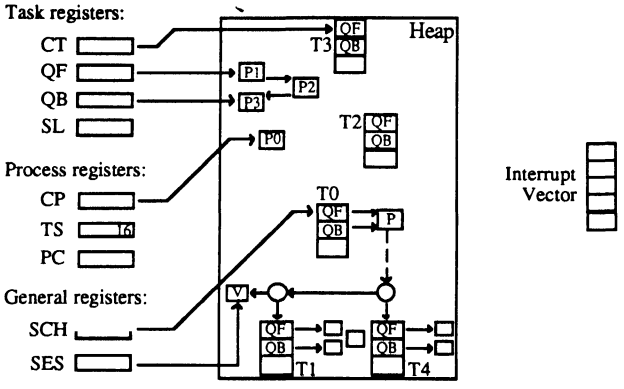


Figure 5.6 Extended Flat Parlog Machine (eFPM) architecture.

Figure 5.6 represents the data structures of the eFPM. Five tasks are represented in this figure. One of these, labelled T3, is the current task; its fields are buffered in the task registers. T2 is deadlocked: its active queue is empty. T0 is the scheduler task; its single process (P) is the scheduler. The scheduler's scheduling structure is currently empty. However, two tasks (T1 and T4) are recorded in the scheduler events stream. (The scheduler possesses a reference to the head of this stream; the SES register points to its uninstantiated tail, the variable V). These will be processed by the scheduler when it is next scheduled.

5.4.4 Implementation of Metacontrol Primitives

The implementation of the kernel language's metacontrol primitives is defined in t of their effect on the data structures of the eFPM. None of these primitives can be applied to the current task; that is, a task cannot control itself. RUN can only be executed by the scheduler.

'TASK'(S,TR): allocates a task record on the heap, returns it as TR and appends it to the scheduler event stream. The new task's process count (PK) is set to one, its message count (MK) to zero, its state (ST) to active and its active queue (QF, QB) to point to the current process (CP). S is recorded as its status variable (SV). The

current process has its task field updated to TR. A new process is then selected from the current (not the new) task.

'SUSPEND'(TR):sets the state field of the task record TR to suspended.

'CONTINUE'(TR): if the state field of the task record TR is suspended, it is set to active and the task record is appended to the scheduler event stream.

'STOP'(TR): sets the state field of the task record TR to aborted.

'PRIORITY'(TR, P): sets the priority field of the task record TR to P and appends the task record to the scheduler event stream.

'RUN'(TR,TSlice): first copies the contents of the task registers to the scheduler's task record. It then records TSlice in the task register SL, loads the other task registers from task record TR, selects a process from task TR's active queue and starts to execute it.

5.4.5 Other Extensions to the FPM

Failure of a process reduction is signalled on the current task's status stream by means of an exception message.

The current task's process count is incremented when a process is created and decremented when a process terminates. This permits termination and deadlock to be detected and reported. If a task's process count is zero, its status variable is instantiated to succeeded to report successful termination. Its task record's state field is also set to terminated and a task switch occurs. If a task's active queue is empty and its message count is zero, but its process count is not zero, a deadlock(N) exception is signalled on its status stream, where N is the process count. Its task record's state field is set to deadlocked and a task switch occurs. A task switch involves saving the contents of the task registers in the current task's task record and then switching to executing the scheduler task.

In the FPM, unification operations check for suspended processes when they instantiate a variable. They move any such processes from the variable's suspension list to the active queue. In the eFPM, an awakened process' task record is consulted to determine which active queue the process is to be appended to. Furthermore, if the awakened process' task was deadlocked, its task record's state field is set to active and the task record is appended to the scheduler event stream. An undeadlock exception is signalled on its status stream. An undeadlock exception is also signalled if a deadlocked task's message count becomes non-zero (see Section 5.5).

Interrupts are handled using the interrupt vector. Interrupts indicating error conditions are signalled on the current task's status stream as exception messages. Timer interrupts cause a task switch if the current task's task timeslice is exhausted.

5.4.6 A Parlog Scheduler

To understand how a Parlog scheduler is used, consider the OS outlined in Section 4.7. This is invoked using a call os_init(Is,N,G). To use a Parlog scheduler, it is instead invoked with a call to the procedure init/4:

```
init(Es, Is, N, G) ← scheduler(Es), call(os_init(Is, N, G), _, _).
```

where Es is the scheduler event stream generated by the kernel. The *initial* task created to execute init/4 creates a *new* task to execute the rest of the OS, whilst it itself continues to execute a procedure scheduler/1. The task record of the initial task is recorded by the eFPM in the SCH register.

A Parlog scheduler can maintain any convenient scheduling structure. As task records are valid Parlog terms, the scheduler can examine their fields. This permits it to verify that they have not been aborted (by examining ST) and to determine their priority (PR). Priorities can be represented and interpreted in any convenient way.

Program 5.4 implements a simple scheduler that maintains a queue of pending tasks, which it schedules using a round robin algorithm. The procedure scheduler/1 is initially invoked with the scheduler event stream as its argument. scheduler/2 then executes as a perpetual process. Each time it is executed, it first adds any new tasks signalled on the scheduler event stream to the tail of its queue (C4-6). It then selects the first task on this queue (C7) and requests the kernel to reduce it (C3). The procedure priority/2 accesses the task's task record (represented as a tuple) to determine its priority (C8). This priority (assumed here to be an integer) is used as its timeslice.

The approach applied here to processor scheduling is quite general and could be used to handle a range of interrupts and other events. For example, the scheduler can be extended to abort tasks when the kernel signals that memory is short (Section 4.2.5). To do this, the scheduler must retain references to the task records of all non-terminated tasks. The kernel is assumed to place a message on the scheduler events stream and to switch to the scheduler when memory is short. The scheduler can then abort tasks to free memory.

```
mode scheduler(Events?), scheduler(Events?, Tasks?), priority(Task?, Priority↑),
     scheduler(Events?, Tasks?, Task?, Time?), sched(Tasks?, Tasks1↑, Task↑, Time↑),
     events(Events?, Events1↑, Tasks?, Tasks1↑), priority(TR?, Priority↑).

scheduler(Es) ← scheduler(Es, [ ]).                    % Initially empty queue.      (C1)

scheduler(Es, Ts) ←                                                                 (C2)
     events(Es, Es1, Ts, Ts1),                         % Add new tasks to queue.
     sched(Ts1, Ts2, Task, Time),                      % Select a task from queue.
     scheduler(Es1, Ts2, Task, Time).                  % Schedule task.

scheduler(Es, Ts, Task, Time) ←                        % Switch to new task.        (C3)
     'RUN'(Task, Time) & scheduler(Es, Ts).

events(Es, Es1, [Task |Ts], [Task |Ts1]) ←             % Find end of queue.         (C4)
     events(Es, Es1, Ts, Ts1).
events([Task |Es], Es1, [ ], [Task |Ts1]) ←            % Add new events to queue.   (C5)
     events(Es, Es1, [ ], Ts1).
events(Es, Es, Ts, Ts) ← var(Es) : true.               % No more pending events.    (C6)

sched([Task | Ts], Ts, Task, Time) ← priority(Task, Time).  % Select 1st task in queue. (C7)

priority({QF,QB,PK,MK,SV,ST,PR}, PR).          % Select priority from task record. (C8)
```

Program 5.4 Parlog task scheduler.

5.5 Distributed Unification

Assume that a kernel language implementation such as that described in Sections 5.3 and 5.4 exists on each node in a multiprocessor. Each node is thus capable of reducing Parlog processes and of creating, monitoring and controlling uniprocessor tasks. Recall that it is assumed herein that nodes in a multiprocessor do not share memory but can communicate by message passing (Section 1.2.1). If the kernel language is to be used to implement multiprocessor operating systems, its implementation must be extended to support:

distributed unification: to permit processes located on different nodes to communicate by unifying shared variables;

distributed metacontrol: to permit the monitoring and control of tasks distributed over several nodes;

global processor scheduling: to permit the allocation of multiprocessor resources to tasks and to processes within tasks.

This section discusses *distributed unification*. Section 5.5.1 describes kernel mechanisms that support distributed unification in Parlog. These can easily be incorporated in the FPM. This material is based on work by Taylor *et al.* [1987b] on multiprocessor implementation of the related language FCP. Section 5.5.2 extends Taylor's work by providing unification algorithms adapted for Parlog's operational semantics. These are compared and contrasted with Taylor's FCP algorithms in Section 5.5.3.

Sections 5.5.4–5.5.7 describe distributed unification algorithms that permit termination and deadlock detection within Parlog tasks. These problems are not addressed by Taylor *et al.*

Distributed metacontrol and *global processor scheduling* are considered in Chapter 6.

5.5.1 Kernel Support for Distributed Unification

If Parlog programs are to execute on multiprocessors, processes located on different nodes must be able to share variables. Only a single occurrence of each such shared variable is maintained; other occurrences are represented by remote references. A **remote reference** specifies a node and a location within a node. Each node is assumed to have a unique identifier. For example, in Figure 5.7 a variable X is located on a node named n2 and is represented on nodes n1 and n3 by remote references.

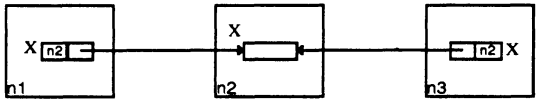


Figure 5.7 Remote references.

The reduction algorithm implemented in the FPM is extended to incorporate **distributed unification algorithms** which generate messages to other nodes when unification encounters terms represented by remote references (**remote terms**). These messages retrieve values or request other nodes to perform unification operations.

These algorithms permit Parlog processes to access terms represented by remote references as if these terms were located locally. They effectively implement a global address space.

Recall that the *test* phase of the FPM's reduction algorithm attempts to reduce a process by performing *tests* (defined by input arguments and guards) on process arguments. If any clause is found to be capable of reducing the process, the *spawn* phase performs *unifications* defined by that clause's output arguments and body primitives. When reduction of a Parlog process leads to suspension the process is suspended on variables for which tests in suspending clause(s) require values. This ensures that a suspended process is not selected for reduction again until one of the variables on which it suspended is given a value.

It is useful to distinguish between strict and non-strict tests. Most Parlog tests are **strict**: they suspend until all input arguments are instantiated and then succeed or fail. For example, data or atomic. A few, such as == and =/=, can *sometimes* proceed when their input arguments are variables. These are said to be **non-strict**. Recall that a call X==Y (Section 3.2.3) succeeds if its two arguments are syntactically identical; this includes the case when they are identical variables. A call X==Y can thus succeed when its arguments are both variable, if they are the *same* variable.

In a multiprocessor, both *tests* and *unifications* can encounter remote references. A *test* operation applied to a remote reference is made to suspend as if the remote reference denoted a variable. If the process itself suspends, messages are generated to request the values of any remote references encountered in suspending clauses. The process is suspended on these remote references; it will thus be awoken when a requested remote value is returned. (The term 'value' is defined below).

Requests generated following strict and non-strict tests are distinguished. As a strict test cannot proceed until a remote term is instantiated, it is not necessary to return the value of a remote term required by a strict test until it is non-variable. A non-strict test, on the other hand, needs to know if the remote term is a variable and, if so, what is its location. (For example, a call X==Y must succeed if X and Y are remote references to the same variable). The value of a remote term must thus be returned immediately when required by a non-strict test, even if it is variable.

A *unification* operation applied to a remote reference can be translated into a message requesting the node on which the remote term is located to perform the unification operation.

The principal extensions to the kernel required to support this extended reduction algorithm consist of a mechanism for receiving and processing incoming messages

(potentially modifying FPM data structures) and a mechanism for transmitting outgoing messages. Figure 5.8 illustrates this extended architecture.

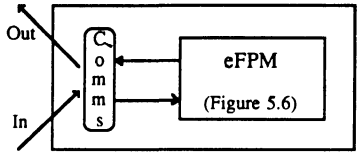


Figure 5.8 Kernel support for distributed unification.

5.5.2 Distributed Unification Algorithms

The extensions to Parlog's reduction algorithm required to support distributed unification are encoded in two distributed unification algorithms. The *read* algorithm is used to retrieve remote terms encountered during the *test* phase of Parlog reduction. This is invoked when a process suspends and is found to require remote values. The *unify* algorithm is used to perform unification operations involving remote terms. This is invoked during the *spawn* phase when unification operations encounter remote references.

Distributed unification algorithms are defined in terms of:

- when messages are generated, and
- what messages are generated, and
- how messages are processed.

Messages are generated as a result of test and unification operations or whilst processing messages. Messages are processed by the communication mechanism of the extended FPM. This may generate further messages and/or modify FPM data structures. Nodes are assumed to alternately perform reduction attempts and process messages; this ensures that FPM data structures are not accessed whilst in inconsistent states.

Messages are represented as structured terms: read(T,F), unify(X,T,Y), etc. The first component of a message (T, X, etc.) is always a remote reference (that is, a {node, location} pair) representing the *destination* of the message.

When a message is generated, it is sent to the node referenced by its first component. A node receiving a message examines the location referenced by this component. If this is a remote reference, the message is forwarded: this dereferences

chains of remote references. Otherwise, the node can proceed to process the message. Dereferencing of remote reference chains is assumed in the following descriptions and is not mentioned explicitly.

A chain of remote references may lead a message back to its source node, making an apparently remote operation a local operation [Taylor *et al.*, 1986b]. For clarity of presentation, the descriptions that follow ignore such special cases. Only minor modifications to the algorithms are required to deal with them.

5.5.2.1 The read Algorithm

If a *strict* test requires the value of a remote term, a read message is generated to the node indicated in the remote reference. A node receiving a read message returns the value of a *non-variable* term immediately using a value message. The value of a *variable* is not returned until the variable is bound. A **broadcast note** is attached to the variable to record the pending request. It is thus necessary to check for broadcast notes when binding variables. Pending requests are responded to with value messages.

The value of a term is defined to be the scalar value of a constant, the top level of a structure (including constant subterms and remote references to other subterms) and a remote reference to a variable.

A value message copies the value of a term from one node to another. (This value is generally a non-variable term, but can be a remote reference to a variable: see below). A node receiving a value message replaces the remote reference with the value and awakens any processes suspended waiting for it. Subsequent accesses to a non-variable value do not require communication. This copying of non-variable terms is possible because of the single-assignment property of Parlog variables. Once a Parlog variable is instantiated, it cannot be modified. Its value can thus be duplicated on several nodes without concern for the consistency of the copies.

If a *non-strict* test requires the value of a remote term, a ns_read message is generated to the node indicated in the remote reference. If the remote term is *non-variable*, its value is returned using a value message, as before. If it is a *variable*, a broadcast note is attached to the variable *and* a remote reference to the variable is returned in a ns_value message. The node that initiated the request can then replace the initial remote reference (which may have been the head of a reference chain) with this direct remote reference. Non-strict tests that required the remote term may then be repeated. If they still suspend, there is no need to request the value of the remote term again. The broadcast note attached to the remote variable means that its value will be

returned as soon as it becomes bound.

Note that a read message is only generated for the first process to suspend on a particular remote reference. A ns_read message is generated for the first process to suspend on a particular remote reference because of a non-strict test.

The read and ns_read messages have the form:

```
read(To, From)
ns_read(To, From)
```

where To is a remote reference to the remote term (that is, a representation of its node and location) and From represents the node and location of the original remote reference.

The value and ns_value messages have the form:

```
value(From, Value)
ns_value(From, Value)
```

where Value is the value of the remote term and From specifies the location of the original remote reference.

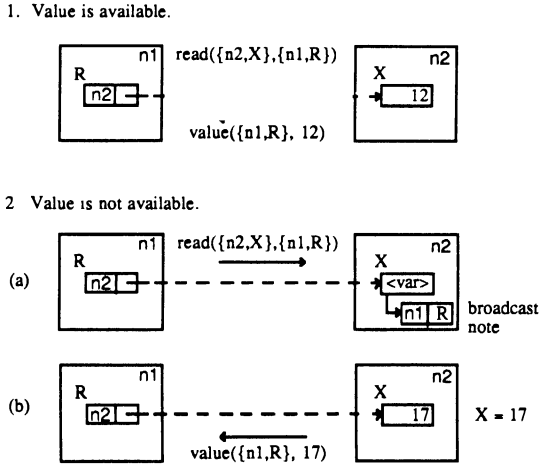


Figure 5.9 The *read* distributed unification algorithm: a strict test.

Figure 5.9 shows two examples of remote reading. In each case, the value of a term represented by a remote reference (represented here as {n2, X}: that is, location X on node n2) is required by a strict test. In the first example, the remote term is available and is returned immediately using a value message. In the second case, the

remote term is not available: location X is a variable. A broadcast node is therefore associated with the variable (a). When this variable is instantiated, its value is returned using a value message (b).

Note that value messages are generated to nodes recorded in a variable's broadcast list when the variable is bound, even if it is only bound to another variable. This 'non-strict' generation of value messages can result in unnecessary communication, as if strict tests require the value of the remote term, further read messages must be generated to request the value of the new variable. However, this avoids the need (a) to concatenate broadcast lists when unifying two variables and (b) to distinguish broadcast notes representing 'strict' and 'non-strict' requests. ('Non-strict' requests must be responded to when the variable is bound to another variable; 'strict' requests only need to be responded to when the variable is instantiated).

Figure 5.10 illustrates the use of ns_read and ns_value messages. In (a), a non-strict test U==V encounters two remote references. Although these indicate different locations on node n2 they in fact refer to the same variable, X. The test U==V initially suspends and ns_read messages are generated. Reference chains are dereferenced and 'direct' remote references returned to node n1. These replace the original remote references. In (b), the test U==V is repeated and, as U and V are now identical remote references, succeeds.

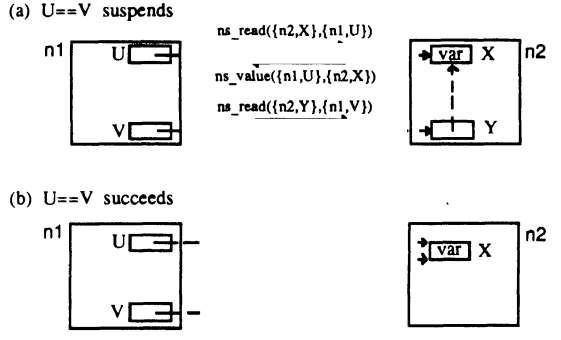


Figure 5.10 The *read* distributed unification algorithm: a non-strict test.

Note that the ns_read requests made to retrieve the value of the remote variable referenced by U and V mean that although the test U==V has succeeded, a broadcast note is associated with this variable and its value will be propagated to node n1 when it becomes bound. It may be worthwhile to 'cancel' this broadcast note so as to avoid this potentially unnecessary communication.

5.5.2.2 The unify Algorithm

Recall that unification is a recursive matching procedure that attempts to make two terms identical, binding variables in either term if necessary (Section 2.5.1). Unification operations are performed during the spawn phase of Parlog reduction. If a unification operation encounters a remote reference, a unify message is generated to request nodes on which remote term(s) are located to continue unification. Failure of such a remote unification operation is signalled to the task that initiated it by a failure message. This permits a failure exception message to be signalled on the status stream of the metacall that initiated the task.

Consider a unification operation X=Y. If one of the terms X or Y is represented by a remote reference, a unify message is generated to the node referenced by the remote reference. A unify message carries the value of the other term to be unified to the node on which the remote term is located.

If both terms to be unified are remote, a unify1 message containing remote references to both terms is dispatched to the node on which the first is located; a node receiving such a message forwards a unify message containing the value of that term to the node on which the second term is located.

A unify message has the form:

```
unify(X, T, Y)
```

where X is a remote reference to a term, T is a remote reference to (the task record of) the task which initiated the unification operation and Y is the value of a term. A unify1 message has the same form: unify1(X, T, Y), but both X and Y are remote references to terms.

A failure message has the form:

```
failure(T, X, Y)
```

where T specifies the location of the task which initiated the unification operation X=Y that has resulted in failure, and X and Y are the values of the terms that could not be unified.

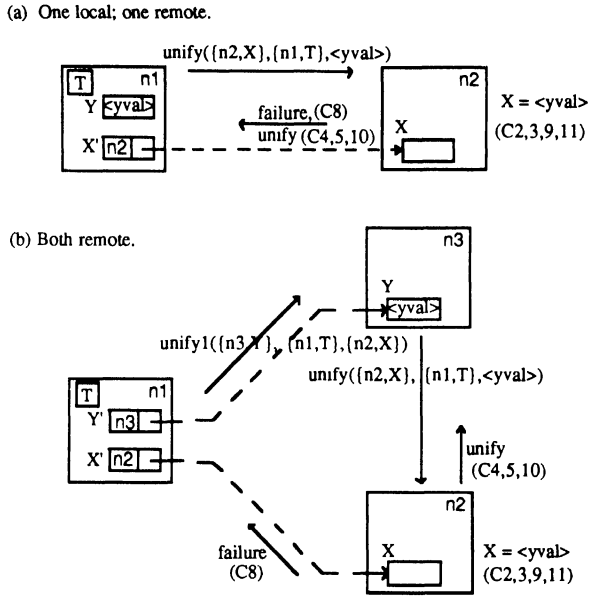


Figure 5.11 The *unify* distributed unification algorithm.

Figure 5.11 illustrates the messages that may be generated by the *unify* algorithm during spawn phase remote unification if one or both terms to be unified are remote.

In (a), only one of the terms to be unified, X, is represented by a remote reference. A unify message is generated to node n2. This contains remote references to X and to the task record of the task in which the unification operation occurs (T), plus the value of the other term, Y (<yval>).

In (b), both terms to be unified are represented by remote references. A unify1 message is generated to n3, the node on which one of these terms, Y, is located. This carries remote references to X and Y. Node n3 receives the unify1 message, determines the value of Y (<yval>) and forwards a unify message to node n2. As in (a), this contains remote references to X and to the task record T, plus the value of the other term, Y (<yval>).

In both cases, node n2 receives a unify message and performs the actual unification operation. The unification algorithm applied, and the function of the additional failure and unify messages represented in the figure, is made clear below. Clause numbers in the figure (Ci) refer to a Parlog specification of the unification algorithm, also presented below.

A node receiving a unify message processes it using the algorithm specified by the Parlog procedure unify/5 (Program 5.5). unify(X,T,Y,Nx,Ns) reads: Ns is the messages generated when a message unify(X,T,Y) is processed on a node with identifier Nx. Recall that when a message unify(X,T,Y) is processed, X references a local value, while Y is a remote value: a non-variable term or a remote reference to a variable.

In Figure 5.11 (a) and (b), node n2 receives a unify message:

```
unify({n2,X}, {n1,T}, <yval>)
```

It invokes the unify algorithm as follows:

```
?- unify(X, {n1,T}, <yval>, n2, Ns)
```

to determine the messages (Ns) that it must generate to unify local term X with the value of the other, remote term, Y (<yval>). X can reference a constant, structure or variable. <yval> is a constant, the top level of a structure, or a remote reference to a variable. If it is a remote reference to a variable, it has the form {n1,Y} in (a); {n3,Y} in (b).

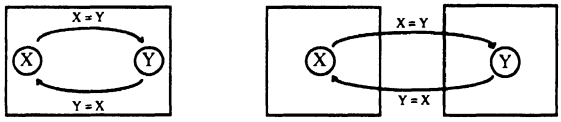
```
mode  unify(X?, T?, Y?, Nx?, Messages↑), order_check(X?, T?, Y?, Nx?, Messages↑),
      unify_structure(X?, T?, Y?, Nx?, Messages↑), node(RemoteRef?, Node↑).

unify(X, T, Y, Nx, [ ]) ← atomic(X), atomic(Y), X == Y : true.   % Success.                (C1)
unify(X, T, Y, Nx, [ ]) ← var(X), atomic(Y) : X = Y.             % Instantiate X.          (C2)
unify(X, T, Y, Nx, [ ]) ← var(X), structure(Y) : X = Y.          % Instantiate X.          (C3)
unify(X, T, Y, Nx, [unify(Y,T,X)] ) ← var(Y), atomic(X) : true.  % Repeat.                (C4)
unify(X, T, Y, Nx, [unify(Y,T,X)] ) ← var(Y), structure(X) : true. % Repeat.             (C5)
unify(X, T, Y, Nx, Ns) ←                                         % Recursively unify structures. (C6)
      structure(X), structure(Y) : unify_structure(X, T, Y, Nx, Ns).
unify(X, T, Y, Nx, Ns) ← var(X), var(Y) : order_check(X, T, Y, Nx, Ns);  % Note ';'     (C7)
unify(X, T, Y, Nx, [failure(T,X,Y)] ).                           % Default: signal failure. (C8)

order_check(X, T, Y, Nx, [ ]) ← node(Y, Ny), Nx < Ny : X = Y.    % X = remote ref.       (C9)
order_check(X, T, Y, Nx, [unify(Y,T,X)]) ←                       % Order check failed: repeat. (C10)
      node(Y, Ny), Nx > Ny : true;                               % Note ';'
order_check(X, T, Y, Nx, [ ]) ← X = Y.                           % Same node.            (C11)
```

Program 5.5 The *unify* distributed unification algorithm.

Unification of two constants either succeeds (C1) or fails (C8). A local variable X is bound to a constant or structure Y (C2,3). An attempt to unify a local constant or structure X with a remote variable Y leads to the forwarding of a unify message

containing the value of X to the node on which Y is located (C4,5; and see Figure 5.11). (structure(X) reads: X is a structured term: a tuple or a list). This permits the variable Y to be bound to the value of the constant or structure X. To unify two structures, subterms are recursively unified using the same algorithm unification; this may generate further unify messages (C6). These are not pictured in Figure 5.11. (unify_structure(X,T,Y,Nx,Ns) reads: Ns is the messages generated when a structure X is unified with a structure Y on a node Nx). Unification of two variables requires an order check — explained below — to avoid the creation of circular remote references (C7). Unification of any other combination of terms is signalled as failure (C8). Failure is signalled to the task which initiated the unification operation by a failure message (see Figure 5.11).

In logic programming systems, circular references can be created if a variable X can be bound to a variable Y at the same time as Y is bound to X, as illustrated in Figure 5.12. This problem can be avoided on uniprocessors by using pointer comparison to ensure that variable to variable bindings are only created in a certain direction (from low address to high address, for example). A similar technique can be used on multiprocessors. An ordering is defined on node identifiers. An **order check** compares node identifiers when variables located on different nodes are unified. Bindings are only permitted from a node of lower identifier to a node of higher identifier.



(a) Uniprocessor (b) Multiprocessor

Figure 5.12 Circular references.

The procedure order_check (Program 5.5: C9-11) applies this technique. A simple numerical ordering on integer-valued node identifiers is assumed here. order_check(X,T,Y,Nx,Ns) reads: Ns is the messages generated when a variable X located on a node Nx is unified with a variable represented by a remote reference Y. node(Y,N), not defined, reads: Y is a remote reference and refers to node N. If the ordering constraint is violated, the unify message is forwarded to the other node (C10). This causes the unification operation to be repeated in the opposite direction and thus creates the binding in the correct direction. Otherwise, X is bound to a remote reference to Y (C9), unless both variables are located on the same node (C11).

Finally, it should be noted that value messages are also generated if variables to which broadcast notes are attached are bound as a result of unification operations. These are not shown in Figure 5.11.

5.5.3 Comparison with Taylor *et al.*'s Distributed Unification Algorithm

Both the kernel mechanisms used to implement distributed unification (Section 5.5.1) and the read/value protocol used in the *read* distributed unification algorithm (Section 5.5.2.1) are based on work by Taylor *et al.* [1987b] on the multiprocessor implementation of FCP. The ns_read/ns_value protocol (Section 5.5.2.1), and the *unify* algorithm (Section 5.5.2.2), are new. These exploit differences between Parlog's and FCP's operational semantics to provide what appear to be more efficient distributed unification algorithms.

FCP uses general unification rather than input matching to determine whether a clause can reduce a process (see Section 8.2.1). Certain unification operations can be distinguished as tests; if these encounter remote references, the read/value protocol described in Section 5.5.2.1 is used to retrieve their values. Otherwise, the use of general unification means that a clause may be required to successfully perform two or more unification operations before it can be selected to reduce a process. These must be performed *as an atomic action*: if any one fails, the others must not occur. To provide this atomicity, the FCP kernel supports *variable migration*. The distributed unification algorithm fetches all variables that are to be bound locally before performing reduction. To prevent livelock when several nodes require the same variables, variables fetched in this way are locked; this in turn requires a deadlock prevention scheme. Starvation is possible. Once all variables required by a process have been fetched locally, reduction can occur without further communication. Any binding performed during a reduction attempt is recorded and undone if reduction fails. As reduction attempts and message processing are alternated, these 'unsuccessful bindings' are not visible to other processes. Taylor's distributed unification algorithm does not incorporate a ns_read/ns_value protocol. Non-strict tests can presumably be implemented by migrating variables to the node where they are to be tested.

In Flat Parlog, on the other hand, variables are only tested prior to reduction. Unification is performed after reduction in a number of independent operations. Unification operations involving remote terms are requested using *unify* messages and variables do not migrate. Alternating message processing and process reduction at

nodes provides mutual exclusion when binding individual variables. As processes do not compete for resources (variables), deadlock, livelock and starvation cannot occur.

Experimental studies are required to quantify the run-time costs associated with the FCP and Parlog distributed unification algorithms; these have not been performed. Intuitively, it would seem that the Parlog algorithms are less expensive, as variables do not need to be migrated before being bound, and variable locking and deadlock detection are not required.

5.5.4 Termination and Deadlock Detection

It can be useful to detect two changes in the state of a Parlog task:

termination:	the process pool representing the task becomes empty.
deadlock:	the process pool is not empty, but all processes become suspended due to dataflow constraints.

It is then possible to signal termination or deadlock on the status stream of the metacall that initiated the task.

As noted in Section 5.4, a uniprocessor implementation of Flat Parlog can maintain a process count for each task in order to detect *termination*. A task's process count is incremented when a new process is created. It is decremented when a process terminates. If a process count reaches zero, the corresponding task has terminated.

A uniprocessor implementation of Flat Parlog can use its active queue to detect *deadlock*. An empty active queue (and a non-zero process count) signify deadlock.

Now consider a Parlog task executing on a single node in a multiprocessor. If processes in such a task do not share variables with processes located on other nodes, the mechanisms just described can be used to detect termination and deadlock. If they do, however, these mechanisms are no longer adequate.

Execution of a process that shares variables with processes located on other nodes may generate unify messages to request remote unification operations. As these operations may result in failure, a task cannot be said to have terminated successfully until its process count is zero *and* it is known that all remote unifications that it has initiated have terminated successfully.

Execution of a process that shares variables with processes located on other nodes may generate read messages to request remote values. Processes suspend until a reply to a remote read is received. A task cannot be said to have deadlocked until its active queue is empty *and* it is known that all remote read requests have either completed or

are suspended waiting for data.

To avoid these problems, alternative *read* and *unify* algorithms are defined. These can be used when termination and/or deadlock detection is required in uniprocessor tasks that are known to share variables with tasks located on other nodes. The algorithms are **symmetric**: that is, they acknowledge messages to indicate that (for example) unifications have terminated or remote reads have suspended. This permits the use of message counting to detect termination and deadlock. The *message count* field (MK) in the task record (Section 5.4.3) is used for this purpose.

Three additional algorithms are defined: *s_unify*, *s_read* and *d_read*. *s_unify* differs from the *unify* algorithm in that it acknowledges successful unifications. *s_read* differs from the *read* algorithm in that it acknowledges read messages for which a broadcast note is generated. The *d_read* algorithm acknowledges both read and value messages.

The various distributed unification algorithms vary in their run-time costs and the metacontrol functions that they support. Different tasks are permitted to use different combinations of these algorithms; which combination is to be used can be specified when the task is created. Four useful combinations can be identified:

C1	*unify, read*	supports neither termination nor deadlock detection.
C2	*s_unify, read*	supports termination detection.
C3	*s_unify, s_read*	also supports deadlock detection in uniprocessor tasks.
C4	*s_unify, d_read*	also supports deadlock detection in multiprocessor tasks.

Combination C1 permits efficient distributed unification in tasks for which neither termination nor deadlock detection is required. For example, system services, which are not expected to terminate but which are represented as separate tasks to permit prioritized scheduling or to localize component failure.

Combination C2 permits termination detection at the cost of some additional communication. This can be used for application programs which have been debugged and are assumed to be deadlock free. Finally, combinations C3 and C4 can be used when deadlock detection is required in uniprocessor or multiprocessor tasks.

5.5.5 Symmetric Distributed Unification Algorithms

5.5.5.1 The s_unify Algorithm

The symmetric *s_unify* algorithm acknowledges successful remote unifications. This permits termination detection in uniprocessor tasks. It differs from the *unify* algorithm (Section 5.5.2.2) in three respects:

- It uses s_unify and s_unify1 messages rather than unify and unify1 messages.
- It acknowledges successful remote unifications. A node receiving a s_unify message processes it as it would a unify message, but acknowledges successful completion of the remote unification operation using an ack message.
- It does not perform remote unification operations when they involve two structures (Program 5.5: C6). Instead, it generates a structure message to return both structures to the node which initiated the unification. This can then initiate new unification operations, one per structure element. This avoids a need for the complex kernel mechanisms that would be required to detect termination of the recursive unification of two remote structures.

```
mode  s_unify(X?, T?, Y?, Nx?, Messages↑), s_order_check(X?, T?, Y?, Nx?, Messages↑),
      node(RemoteRef?, Node↑).

s_unify(X, T, Y, Nx, [ack(T) ]) ← atomic(X), atomic(Y), X == Y . true. % Success.      (C1)
s_unify(X, T, Y, Nx, [ ack(T)]) ← var(X), atomic(Y) : X = Y.        % Instantiate X.  (C2)
s_unify(X, T, Y, Nx, [ack(T) ]) ← var(X), structure(Y) : X = Y.     % Instantiate X.  (C3)
s_unify(X, T, Y, Nx, [unify(Y,T,X)] ) ← var(Y), atomic(X) : true.   % Repeat.         (C4)
s_unify(X, T, Y, Nx, [unify(Y,T,X)] ) ← var(Y), structure(X) : true. % Repeat.       (C5)
s_unify(X, T, Y, Nx, [structure(T,X,Y)]) ←                          % Signal structure. (C6)
      structure(X), structure(Y) . true.
s_unify(X, T, Y, Nx, Ns) ← var(X), var(Y) : order_check(X, T, Y, Nx, Ns);  % Note ';'  (C7)
s_unify(X, T, Y, Nx, [failure(T,X,Y)] ).                % Default: signal failure.  (C8)

s_order_check(X, T,  Y, Nx, [ ack(T)]) ← node(Y, Ny), Nx < Ny : X = Y.  % remote ref. (C9)
s_order_check(X, T, Y, Nx, [unify(Y,T,X)]) ←          % Order check failed: repeat. (C10)
      node(Y, Ny), Nx > Ny : true;                    % Note ';'
s_order_check(X, T, Y, Nx, [ ack(T)]) ← X = Y.        % Same node.               (C11)
```

Program 5.6 The *s_unify* distributed unification algorithm.

The algorithm used to process a s_unify message is defined by Program 5.6. Compare this with Program 5.5. Note the ack message generated in C1-3,9,11 and the structure message generated in C6.

A node increments its process and message counts when it generates a s_unify message and decrements them when it receives a failure or ack message. A process count of zero then signifies that both all a task's processes *and* all remote unifications that it has initiated have terminated. This is termination. (The message count is modified to permit deadlock detection: see below).

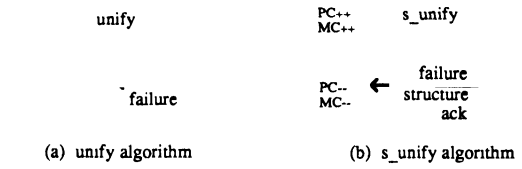


Figure 5.13 The *unify* and *s_unify* distributed unification algorithms.

Figure 5.13 illustrates the messages generated by the *unify* and *s_unify* algorithms. The *unify* algorithm generates a unify message, which *may* receive a failure response. The *s_unify* algorithm generates a s_unify message, which *always* receives a failure, structure or ack response. In this figure, PC++, MC--, etc. indicate the increment and decrement of the process and message counts of the task that initiated the unification operation.

s_unify messages have the same form as unify messages (Section 5.5.2.2):

```
s_unify(X, T, Y)
s_unify1(X, T, Y)
```

The ack and structure messages have the form:

```
ack(T)
structure(T, X, Y)
```

where T specifies the location of the task which initiated the remote unification operation and X and Y are the two structures that are to be unified.

5.5.2 The s_read Algorithm

For brevity, the descriptions of symmetric *read* algorithms that follow only treat the read/value protocol used to retrieve remote values required by strict tests. The

ns_read/ns_value protocol used to retrieve remote values required by non-strict tests (Section 5.5.2.1) can be modified in a similar way.

The symmetric *s_read* algorithm acknowledges remote read requests. This permits deadlock detection in uniprocessor tasks. It differs from the read algorithm (Section 5.5.2.1) in three respects:

- It generates s_read rather than read messages to request remote values.
- A node receiving a s_read message acknowledges it with an ack message if the value requested is not available and a broadcast note must be created.
- If the value is available, it is returned in a svalue rather than a value message message. (If a broadcast note is created, then the value is returned when it becomes available in a value message, as in the *read* algorithm).

A node increments a task's message count when it transmits a s_read message; it decrements it when it receives a svalue or an ack message. The svalue message effectively combines the acknowledgement to the read (ack) with the message that returns the value (value). This optimization reduces communication when a value is immediately available.

When the *s_read* algorithm is used in conjunction with the *s_unify* algorithm, a message count of zero signifies that all remote unifications that a task has initiated have terminated and that all remote reads it has initiated have either completed or resulted in suspension. If the active queue is also empty, indicating that the task has no active processes, the task is deadlocked.

Figure 5.14 illustrates the messages generated by (a) the *read* algorithm, (b) the *s_read* algorithm when a remote value is available and (c) the *s_read* algorithm when a remote value is not available. In (b) and (c), MC++ and MC-- denote the increment and decrement of the message count of the task that initiated the s_read operation.

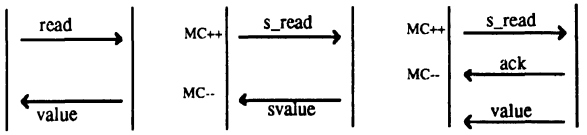


Figure 5.14 The *read* and *s_read* distributed unification algorithms.

The s_read and svalue messages differ from the read and value messages in incorporating the location of the task that initiated the read, T. They have the form:

```
s_read(From, T, To)
svalue(From, T, To )
```

5.5.3 The d_read Algorithm

A second symmetric *read* algorithm, *d_read*, acknowledges both read and value messages. This permits deadlock detection in distributed tasks programmed in Parlog using two or more uniprocessor tasks (see Section 6.2). It differs from the *s_read* algorithm just described in two respects:

- It generates d_read rather than read messages.
- Values that are not immediately available are returned by a dvalue rather than a value message when they become available. (The broadcast note generated for a d_read request must indicate that a dvalue message is to be generated).

A dvalue message includes a remote reference to the task that performed the unification operation that bound the remote variable. (This is *not* the same as the task that generated the d_read message; it is located on a different node). A dvalue message is acknowledged by the node that receives it with an ack message; this is directed to the task that bound the variable.

A node increments a task's message count when it transmits a d_read or a dvalue message; as before, it decrements it when it receives an svalue or an ack message. The *d_read* algorithm can thus modify the message counts both of tasks that require values (those that generate d_read messages) and of tasks that generate values (those that instantiate variables required by d_read requests).

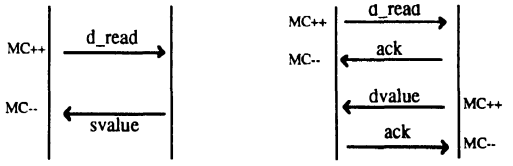
When the *d_read* algorithm is used in conjunction with the *s_unify* algorithm, a message count of zero signifies that all read, value and unify messages that a task has generated have been received.

Figure 5.15 illustrates the messages and message count changes when the *d_read* algorithm is used and (a) a value is available; (b) a value is not available.

The d_read and d_value messages have the form:

```
d_read(To, T, From)
dvalue(From, T, R, Value )
```

To and From specify the node and location of the value to be read and the original remote reference, respectively. Value is the value of the remote term. Ť specifies the location of the task that initiated the read operation. R is a remote reference to the task that generated the dvalue message by supplying the value To. This is used when acknowledging the dvalue message. The acknowledgement is the message: ack(R).



(a) d_read algorithm: data available

(b) d_read algorithm: data not available

Figure 5.15 The *d_read* distributed unification algorithm.

5.5.6 Complexity of Distributed Unification

A simple example illustrates the use of distributed unification algorithms and motivates some observations on their complexity.

Distributed unification allows the use of Parlog to express distributed algorithms. For example, consider the program:

```
mode producer(Xs↑,S?), consumer(Xs?)

producer([X |Xs], synch) ← producer(Xs, X).

consumer([X |Xs]) ← X = synch, consumer(Xs).
```

and the query:

```
?- producer(Xs, synch), consumer(Xs).
```

The program implements synchronous communication between producer and consumer. producer communicates a variable X to consumer by instantiating the shared variable Xs to a list structure containing the variable. It then suspends until X is bound. consumer acknowledges each such 'communication' by instantiating the variable to synch. This permits producer to generate a further communication.

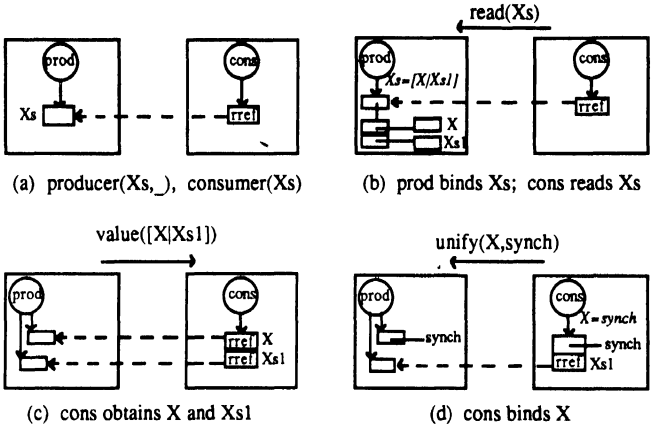


Figure 5.16 Distributed unification.

Assume that producer and consumer are located on different nodes. Figure 5.16 illustrates the messages generated when the conjunction is executed by a Parlog implementation using the *unify* and *read* distributed unification algorithms. In (a), the initial situation is represented. It is assumed that the variable Xs is located on the same node as producer; consumer thus possesses a remote reference to this variable. In (b), producer generates a value for Xs (say [X |Xs1]) on its own node; when consumer attempts to read the shared variable Xs, a read message is generated to retrieve that value. In (c), a value message returns the list structure generated by producer to consumer; this contains remote references to X and Xs1. In (d), consumer can now unify X with the constant synch. As X is a remote reference, a unify message is generated to producer.

The Parlog implementation uses *three* messages to send and acknowledge a 'communication'. Two messages are required to 'read' the original value; if the *s_read* or *d_read* algorithms were used, three or four would be required if the value was not immediately available. One message is required to 'write' the value synch to the variable X; two would be required if the *s_unify* algorithm were used.

In contrast, if this algorithm were to be implemented in a language with explicit *send* and *receive* primitives, *two* messages would be required for each 'communication': one to send it and one to acknowledge it.

Two points can be made:

- The distributed unification algorithms presented here are *in general* optimal in their communication complexity. That is, O(N) messages are required to communicate O(N) values between nodes. ('In general', because when unifying two variables, order checks may cause additional communications. This is however a special case, as variables rather than values are involved).
- The distributed unification algorithms presented here are *lazy*: a value must be requested by a reader before it is transmitted. This is why three messages are required to communicate two values. There is considerable scope for optimizations that eagerly propagate values when readers are known to require them.

This example also illustrates a useful optimization that can be made in an implementation of the *unify* algorithm. If a remote reference is encountered during a unification operation, a unify message is generated, as in Figure 5.16 (d). The remote reference can *immediately* be replaced with the term with which the remote term is to be unified. (In Figure 5.16 (d), the string synch). Subsequent references to that term need not therefore generate communications.

5.5.7 Discussion

Distributed unification algorithms permit processes located on different nodes in a multiprocessor to communicate by unifying shared variables. A number of such algorithms have been defined. These vary in their run-time costs and the metacontrol functions that they support (Section 5.5.4). Different tasks are permitted to use different combinations of these algorithms; which combination is to be used can be specified when the task is created. Experimental studies are required to quantify the run-time costs associated with the different algorithms; these have not been performed.

The distributed unification algorithms are simple. There are three basic message types — read, value and unify — plus three acknowledgement messages: failure, ack and structure. As noted in Section 5.5.6, these algorithms are in general optimal in their communication complexity. Except when order checks fail when unifying variables, there is no 'hidden communication': reading or writing a remote value involves a small, constant number of messages. This means that users can visualize the communications implied by algorithms and can hence program particular

communication patterns. Nevertheless, there is still scope for optimizations that permit eager propagation of values, so as to reduce the 'small constant' to one in a greater number of cases.

The kernel mechanisms used to support distributed unification are simple. In particular, no information needs to be maintained about a task on any node other than the node on which it is created. Such 'remote information' would be required in the *s_unify* algorithm, to detect termination of remote structure-to-structure unifications. (The information to be maintained here would be the number of outstanding sub-structure unification operations). This complexity is avoided by the structure message, which returns remote structures to the node that initiated the remote unification operation.

The structure message is, strictly speaking, an unnecessary communication. It may thus appear to be a source of inefficiency. However, unification of two structures, though possible in Parlog, occurs rarely in practice. To determine the approximate frequency of such operations, ten Parlog applications (including those analysed in Section 5.2.3) were tested on a uniprocessor Parlog implementation. No structure-to-structure unifications were detected. As remote structure-to-structure unifications are a subset of all structure-to-structure unifications on a multiprocessor, it can be expected that such operations will be extremely rare. The structure message thus appears to be a useful and inexpensive simplification.

5.6 Approaches to Kernel Design

This chapter has presented an approach to the design of a Parlog OS kernel. The kernel provides direct support for a *kernel language*: the flat subset of Parlog, augmented with metacontrol primitives. It is argued that this approach to kernel design represents a useful compromise between conflicting requirements for functionality, simplicity, flexibility and efficiency. Simplicity and flexibility are emphasized, but kernel support for important metacontrol functions maintains efficiency. Nevertheless, different tradeoffs between these requirements can be made. Two different approaches that have been proposed to kernel design are described below.

5.6.1 Logix

The designers of Logix, a programming environment for FCP (see Section 8.2.1), choose to use a flat parallel logic language as a kernel language [Hirsch *et al.*, 1986; Hirsch, 1987]. No metacontrol primitives are supported by the kernel. Instead, metacontrol functions such as termination detection, abortion, etc. are implemented by program transformation, as described in Section 5.2.4.1.

The virtues of this approach are its simplicity and flexibility. The kernel only needs to support a simple language and new metacontrol functions can be programmed without modifying the kernel.

The principal disadvantage of this approach is its lack of direct support for the important systems programming concept of *task*. Much of the strength of parallel logic languages as systems programming languages stems from their linguistic support for systems programming concepts such as processes, concurrency, communication and synchronization. The 'process subpool' or 'task' is an equally important concept. It thus appears natural to support it in a systems programming language. The benefits of doing so are first of all *efficiency* — kernel mechanisms can easily be optimized — and secondly *functionality*. Kernel support for tasks permits efficient implementation of metacontrol functions such as deadlock detection and task scheduling that appear difficult to implement by transformation. Furthermore, as this chapter shows, these benefits can be achieved without sacrificing simplicity or flexibility.

In summary, the Logix approach provides in some respects greater *simplicity* and *flexibility* than that described herein (though FCP is itself a more complex language than Parlog — see Section 8.2.1 — and program transformations can of course also be applied to Parlog). However, it appears to compromise *efficiency* and *functionality*.

It appears likely that some kernel support for metacontrol functions will be required if FCP is to be used as a systems programming language. What form these primitives should take, and whether they can be introduced into the language without compromising its simplicity, are open research questions.

5.7.2 Kernel Language 1

Researchers at the ICOT research centre have adopted a kernel language (called Kernel Language 1) that is quite similar to that described herein (see Section 8.2.2). However, they propose that the metacontrol functions defined by the control metacall be supported directly by the kernel, both on uniprocessors and multiprocessors [Sato

et al., 1987a; Ichiyoshi *et al.*, 1987]. Tasks may execute on any number of nodes; distributed termination detection, control and exception handling are implemented in the kernel.

The chief advantage of this approach is that metacontrol functions can be implemented particularly efficiently. There is no layer of interpretation as in the metacalls implemented in Section 5.2.2 and in the distributed metacalls to be described in Chapter 6. Also, more efficient communication algorithms than those described herein can be used, because tasks are global rather than uniprocessor entities. For example, Rokusawa *et al.* [1988] describe a distributed termination detection algorithm that uses weighted reference counts to avoid acknowledging every remote unification request.

One disadvantage of this approach is the complex kernel support required to implement distributed metacontrol. In particular, as tasks are global entities, they must be identified by a globally unique, symbolic name. A 'task table' must be maintained at each node. This is consulted when processing messages, to determine whether a task already exists on a node. In contrast, the simpler scheme described in this chapter identifies uniprocessor tasks solely by a remote reference to their task record. No task table or table lookup is required.

A second disadvantage is the potential inflexibility of metacontrol mechanisms fixed in the kernel. For example, the control metacall defined in the kernel language has a single status stream. This centralizes the processing of run-time errors generated by a task, when they could be more efficiently handled at individual nodes. The kernel implementation of the control metacall can be extended to provide this functionality; this however adds to its complexity. In contrast, this functionality can be programmed directly in the kernel language described herein (see Section 6.5).

In summary, this approach provides similar *functionality* to the kernel language described herein. It may be expected to provide greater *efficiency*, though further investigation is required to determine whether this is significant. Demerits of the approach are its lack of *simplicity* and, potentially, *flexibility*.

CHAPTER 6.

Multiprocessor Operating Systems

A multiprocessor implementation of the kernel language described in Chapter 5 permits Parlog processes located on the same or different nodes to communicate using shared logical variables. This uniformity means that Parlog programs such as the simple OS presented in Section 4.7 can be executed on a multiprocessor simply by placing processes on different nodes. However, issues such as the monitoring and control of distributed tasks, multiprocessor scheduling, potential inefficiencies due to bottlenecks and excessive communication, and the location of services require special consideration. This chapter considers how these issues can be treated in Parlog.

The first two sections deal with the problem of monitoring and controlling tasks distributed over several nodes. It is shown that distributed metacontrol functions can be programmed in the kernel language described in Chapter 5.

Processor scheduling is a particularly complex problem on multiprocessors. If it is assumed that there are more nodes than tasks and that tasks are to be allocated disjoint sets of nodes, then it may be subdivided into two, still complex problems: that of allocating nodes to tasks and that of mapping processes to nodes. Section 6.3 deals with the problem of mapping Parlog processes to the nodes of a multiprocessor. It is shown that process migration can be programmed in Parlog and that the distributed metacontrol mechanisms presented in Sections 6.1 and 6.2 can be adapted to deal with process migration.

Section 6.4 presents a multiprocessor version of the uniprocessor OS described in Section 4.7. A discussion of problems of efficiency, location of services and bootstrapping follows. The final section shows how the allocation of nodes to tasks can be programmed in Parlog.

6.1 Distributed Metacontrol

The control metacall defined in Chapter 4 encodes the following metacontrol functions:

- termination detection
- exception handling
- deadlock detection
- control: the ability to suspend, resume and abort evaluation
- prioritized scheduling

The kernel language described in Chapter 5 provides metacontrol primitives that implement these functions with respect to *uniprocessor tasks*: tasks executing on a single node. This section shows how these metacontrol functions can be applied to *distributed tasks*: tasks executing on many nodes in a multiprocessor.

It is assumed initially that processes do not migrate between nodes. (Complications due to process migration are considered in Section 6.3). This means that processes cannot exist in transit between nodes. The process pool representing a distributed task can therefore be viewed as a number of process subpools, one per node on which the task is executing. Each such subpool can be executed as a separate uniprocessor task and can hence be monitored and controlled using the kernel language's metacontrol primitives. If additional supervisor processes are provided that coordinate the monitoring and control of these uniprocessor tasks, then the distributed task itself can be monitored and controlled as if it were executing on a single node.

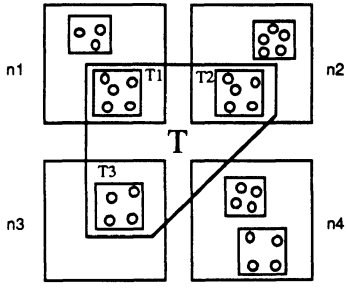


Figure 6.1 A distributed task.

For example, in Figure 6.1, a distributed task T executing on three nodes N1, N2 and N3 consists of three subtasks T1, T2 and T3. To control task T, the subtasks T1,

T2 and T3 must be controlled. To detect termination of the task T, it is necessary to determine that T1, T2 and T3 have terminated.

Consider a query consisting of a conjunction of N goals, G_1, G_2, ..., G_N. This query can be executed as a *uniprocessor* task using the control metacall call/4 (Program 5.1):

call(M, (G_1, . ., G_N), S, C)

where M is a module containing the code for G_1, G_2, ..., G_N.

This query can also be executed as a *distributed* task using the procedure dcall/4 (Program 6.1):

dcall(M, [G_1, ..., G_N], S, C)

dcall/4's second argument is a list of goals G_1, ..., G_N. It executes each of these N goals as a separate uniprocessor task using the kernel language's TASK primitive (C2,4). dcall/4 also creates N local supervisor processes (lsv), one per uniprocessor task, and a single coordinator process (coord). These additional processes coordinate the monitoring and control of the N uniprocessor tasks so as to provide the monitoring and control functions required by the control metacall.

```
mode dcall(Module?, Goals?, Status↑, Control?), dcall1(Module?, Goals?, Left?, Right↑),
     lcall(Module?, Goal?, Status↑, TaskRecord?).

dcall(M, Gs, S, C) ← coord(S, C, L, R), dcall1(M, Gs, L, R).                    (C1)

dcall1(M, [G |Gs], L, R) ← lcall(M, G, S, TR), lsv(S, TR, L, L1), dcall1(M, Gs, L1, R).   (C2)
dcall1(M, [ ], L, L).                                                           (C3)

lcall(M,G,S,C) ← 'TASK'(S,TR) & M@G.                                            (C4)
```

Program 6.1 Distributed metacontrol: top level.

Assume that each {task, lsv} pair executes on a different node in a multiprocessor. (The mechanism used to map tasks and processes to nodes is described in Section 6.4). A call to dcall/4 then creates the process network illustrated in Figure 6.2. Nodes are labelled 1, ..., N.

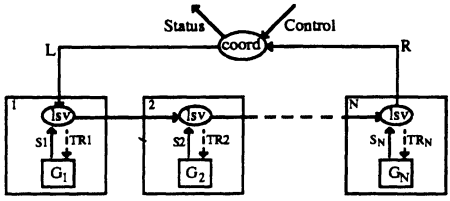


Figure 6.2 Distributed metacontrol.

The local supervisors are connected using stream variables. Each of the N local supervisor lsv_i, $1<i<N$, shares stream variables with supervisors lsv_{i-1} and lsv_{i+1}. This creates a **circuit** that encompasses all the local supervisors, as illustrated in Figure 6.2. Streams to lsv_1 and lsv_N (L and R respectively) provide the coordinator with access to the left and right sides of this circuit. The coordinator also maintains references to the task's status and control streams, S and C.

```
mode coord(Status↑, Control?, Left↑, Right?),  lsv(Status?, TaskRecord?, Left?, Right↑).

coord(S, stop, [stop |L], R) ← coord(S, _, L, R).   % Place stop control on circuit.   (C1)
coord(S, [M |C], [M |L], R) ← coord(S, C, L, R).    % Place other controls on circuit. (C2)
coord(failed, C, [stop |L], [exception(failure,_,_) |R]). % Failure                     (C3)
coord(succeeded, C, L, R) ← L == R : true.          % Circuit closed = success.       (C4)
coord(stopped, C, L, [stop |R]).                    % Entire task stopped.            (C5)
coord(S, C, L, [priority(_) |R]) ← coord(S, C, L, R). % Entire task rescheduled.     (C6)
coord([M |S], C, L, [M |R]) ←                       % suspend, continue, exception.  (C7)
        M=/= stop,  M =/= priority(_),  M =/= exception(failure,_,_) : coord(S, C, L, R).

lsv(S, TR, [stop |L], [stop |L]) ← 'STOP'(TR).      % Control: abort + close circuit. (C8)
lsv(S, TR, [suspend |L], [suspend |R]) ← 'SUSPEND'(TR) : lsv(S, TR, L, R).          (C9)
lsv(S, TR, [continue |L], [continue |R]) ← 'CONTINUE'(TR) . lsv(S, TR, L, R).      (C10)
lsv(S, TR, [priority(P) |L], [priority(P) |R]) ← 'PRIORITY'(TR, P) : lsv(S, TR, L, R). (C11)
lsv(S, TR, [exception(T,G,C) |L], [exception(T,G,C) |R]) ← lsv(S, TR, L, R).       (C12)
lsv(succeeded, TR, L, L).                           % Termination: close circuit.    (C13)
lsv([exception(T,G,C) |S], TR, L, [exception(T,G,C) |R]) ← lsv(S, TR, L, R).       (C14)
lsv([O |S], TR, L, R) ← O =/= exception(_,_,_) : lsv(S, TR, L, R). % Ignore deadlock (C15)
```

Program 6.2 Distributed metacontrol: coordinator and local supervisor.

Program 6.2 implements a local supervisor and coordinator that support all metacontrol functions listed at the start of this section except deadlock detection.

The circuit linking the local supervisors is used for three purposes: to control evaluation, to notify the coordinator of exceptions and to detect global termination.

Control: The coordinator monitors the task's control stream and the right hand side of the circuit. stop, suspend, continue and priority(_) control messages are placed on the left hand side of the circuit (C1,2). A local supervisor receiving such a message controls its task (using kernel language metacontrol primitives) and forwards the message (C8-11). If it is a stop message, the local supervisor terminates (C8); otherwise it recurses to process further messages (C9-11). Recall that metacontrol primitives applied to uniprocessor tasks take immediate effect. When the message placed on the left hand side of the circuit arrives on the right hand side (C5–7), the coordinator hence knows that all subtasks have been controlled. It echoes the control message on the status variable (C5,7). The priority(_) message is not echoed (C6).

Exceptions: A local supervisor generates messages on its output (right-hand) stream when it detects exceptions in its task (C14). It forwards any exception messages received on its input (left-hand) stream (C12). The coordinator thus receives a stream of exception messages on the right-hand side of the circuit. Like call/4, dcall/4 treats process or unification failure in a subtask as fatal. It unifies the task's status variable with a result failed to signal failure, places a stop message on the left-hand side of the circuit to abort evaluation and terminates (C3). Other exception messages are echoed on the task's status stream (C7).

Global termination: When a local supervisor detects termination of its task, it unifies its left- and right-hand streams (C8,13), thus 'closing' its part of the circuit. The coordinator has references to both the leftmost and the rightmost streams in the circuit. These will be identical variables (C4) when all supervisors have detected termination of their task and the entire circuit is thus 'closed'. This programming technique, known as a **short circuit**, is due to Takeuchi [1983].

A call: dcall(M,[G_1,...,G_N],S,C) thus results in the same computation and can be monitored and controlled in the same ways as the call: call(M,(G_1,...,G_N),S,C). However, its evaluation is distributed over N nodes. The only metacall function Program 6.2 does not support is deadlock detection. The implementation of this function is described in the next section.

Program 6.2 can be improved in a number of ways. For example, local supervisors can process certain exceptions locally; this reduces the frequency of communication with the coordinator. Section 6.5 illustrates this optimization.

The problem of executing a single goal rather than a conjunction of goals as a distributed task is dealt with in Section 6.3.

6.2 Distributed Deadlock Detection

A Parlog task is deadlocked when all its processes are suspended due to dataflow constraints (Section 4.2.3). As suspended processes can become active as a result of other process' activity, deadlock detection must verify that *all* processes in a task are *simultaneously* suspended. This is trivial on a uniprocessor as it is possible to have an immediate view of the number of active processes in a task (Section 5.4). Deadlock detection in a distributed task is more complex because a global view of computation state is not immediately available.

Dijkstra *et al.* [1983] describe a termination detection algorithm for distributed computations. This detects global termination (or deadlock) by repeatedly visiting nodes on which a computation is active until it has verified that all are simultaneously *inactive*. The algorithm requires (1) that inactive nodes cannot generate messages, (2) that inactive tasks cannot become active except by receiving messages and (3) that communication is instantaneous.

Each node is assumed to maintain a state: black or white. A node is initially black, may be set to white when visited by the algorithm and reverts to black if any other communication is received. The algorithm visits each node in turn, setting its state to white if it is inactive. If all nodes are inactive when visited, it then revisits all nodes. If all are still white, then it is known that no node has generated or received a communication since the last visit. All nodes are hence known to be simultaneously inactive.

Dijkstra's algorithm can still be used if communication is *not* instantaneous provided that a node is only treated as inactive when it is known both that it is not active *and* that all messages it has generated have been received. This can be verified by causing each message to be acknowledged; if a node has received as many acknowledgements as it has generated messages, it is known that it has no outstanding messages. Dijkstra's algorithm can then be used to verify both that all nodes are simultaneously inactive and that no node has outstanding communications. This is termination, even if communication is not instantaneous.

The *s_unify* and *d_read* distributed unification algorithms (Section 5.5.4) acknowledge the read, unify and value messages necessary for remote unification. In

consequence, when a task that uses these algorithms has an empty active queue and a message count of zero, it is known both that the task has no active processes and that all messages it has generated have been received. If all uniprocessor tasks comprising a distributed task use these algorithms, Dijkstra's algorithm can be used to detect when all tasks are simultaneously inactive *and* have had all their messages received. At this point there can be neither active processes on any node nor messages in transit. Deadlock can hence be confirmed.

Program 6.3 exploits the deadlock(_) and undeadlock status messages provided by the kernel language's uniprocessor metacontrol primitives to implement a variant of Dijkstra's algorithm. This variant visits all nodes in turn but only advances from one node to the next when a node is inactive. It can hence be used to detect deadlock but not to verify whether or not a task is deadlocked. (If a node never becomes inactive, the algorithm never advances).

It is assumed that a distributed task is expressed in terms of a number of uniprocessor tasks and supervisor processes as in Program 6.1 and that each such task uses the *s_unify* and *d_read* distributed unification algorithms.

```
mode coord(Status↑, Left↑, Right?), coord1(Status↑, Left↑, Right?),
     lsv(Status?, Left?, Right↑), lsv(State?, Status?, Left?, Right↑).

coord(S, [yes |L], R) ← coord1(S, L, R).          % Initiate check with 'yes' token. (C1)

coord1([deadlocked |S], L, [yes |R]).             % Deadlock confirmed.              (C2)
coord1(S, [yes |L], [no |R]) ← coord1(S, L, R).   % Not confirmed: repeat check.    (C3)

lsv(S, L, R) ← lsv(active, S, L, R).              % Each subtask initially active.   (C4)

lsv(active, [deadlock(_) |S], L, R) ← lsv(deadlocked, S, L, R).  % State = deadlocked. (C5)
lsv(wait, [deadlock(_) |S], L,[no | R]) ← lsv(white, S, L, R).   % State = white.     (C6)
lsv(Any, [undeadlock |S], L, R) ← lsv(active, S, L, R).          % State = active.    (C7)
lsv(active, S, [Any |L], R) ← lsv(wait, S, L, R).                % State = wait.      (C8)
lsv(deadlocked, S, [Any |L], [no |R]) ← lsv(white, S, L, R).     % State = white.     (C9)
lsv(white, S, [Any |L], [Any |R]) ← var(S) : lsv(white, S, L, R). % Forward token.  (C10)
lsv(active, succeeded, L, R) ← lsv(deadlocked, _, L, R).         % Succeeded.        (C11)
lsv(wait, succeeded, L, [no |R]) ← lsv(white, _, L, R);          % State = white.     (C12)
lsv(Any, [_ |S], L, R) ← lsv(Any, S, L, R).                      % Ignore status.    (C13)
```

Program 6.3 Distributed deadlock detection.

Program 6.3 implements a local supervisor and a coordinator similar to those implemented in Program 6.2. To simplify the presentation, the procedures in Program 6.3 do not provide metacontrol functions other than deadlock detection. The control and task record arguments associated with the equivalent procedures in Program 6.2 are hence omitted. Program 6.3 can easily be extended along the lines of Program 6.2 to provide all metacontrol functions listed in Section 6.1.

Each local supervisor records the state of its task: **deadlocked, active, wait** or **white**. A task is initially **active** (C4). If a deadlock(_) message is received on its status stream it becomes **deadlocked** (C5) until an undeadlock status message is received — in which case it becomes **active** (C7) — or (as described below) a token is received, in which case it becomes **white** (C9). A task also becomes **deadlocked** if it terminates (C11). The state **white** can only change to **active**. This happens if an undeadlock status message is received (C7). Other status messages are ignored (C13).

To check for deadlock, a yes token is passed to the left side of the circuit (C1). Each local supervisor forwards a token received on its left stream if it itself is **white** (C10); changes its state to **white** and forwards a no token if it is **deadlocked** (C9); or changes its state to **wait** if it is **active** (C8). A supervisor in **wait** state forwards a no token and sets its state to **white** if its task deadlocks (C6) or terminates (C12). A token thus circulates round all the monitors until either a yes or no token appears on the right side of the circuit. If a yes token appears, deadlock has been confirmed (C2); otherwise, the process is repeated (C3). Note the use of the var test in C10. By checking that its task's status stream is a variable, a supervisor ensures that there are no pending undeadlock messages and hence that its task is currently deadlocked.

For deadlock to be confirmed, every task must be **white** (and hence continuously deadlocked, with no unreceived messages) for a complete circuit of the token. This implies that all tasks, and hence the task as a whole, are simultaneously deadlocked.

At least two circuits of the token are required to confirm deadlock in a distributed Parlog task. However, each circuit only requires $O(N)$ communications and process reductions, where N is the number of nodes on which the distributed task is executing.

Figure 6.3 illustrates the application of this algorithm.

Note that the deadlock status message generated by Program 6.3 does not indicate the number of processes in the deadlocked task. This information is available at the individual nodes and can easily be accumulated by the yes token as it visits each node. This information is shown to be useful in Section 6.3.3.

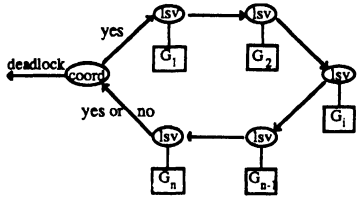


Figure 6.3 Distributed deadlock detection.

6.2.1 An Alternative Approach

Nobuyuki Ichiyoshi [personal communication] has proposed an alternative approach to distributed termination (or deadlock) detection. A count of messages sent and messages received is maintained at each node on which a task is executing. The deadlock detection algorithm then visits each such node, checking that the task is inactive and summing messages sent and messages received. If all nodes are found to be simultaneously inactive *and* the sums of messages sent and messages received by all nodes are equal, then termination has been detected. The message counts indicate that no messages are in transit.

The merit of this scheme, compared with that described herein, is that distributed unification algorithms that do not acknowledge messages (such as *unify* and *read*: Section 5.5) can be used. A disadvantage is that deadlock detection in the uniprocessor tasks comprising a distributed task is not supported. This is shown to be useful in Section 6.3.2.

6.3 Process Migration

In Sections 6.1 and 6.2, it was assumed that queries to be executed as distributed tasks are expressed as conjunctions of goals. Each goal in a conjunction is executed on a different node. This means effectively that a programmer (or compiler) is required to partition a problem into subproblems before it can be distributed. Furthermore, this partitioning is static: processes cannot move between nodes at run-time.

If the run-time behaviour of a program is complex or data-dependent, this static partitioning is unlikely to lead to good processor utilization. It may hence be desirable to redistribute work amongst nodes at run-time. This redistribution may be specified as part of a distributed algorithm (this is referred to here as **mapping**) or may be performed

automatically in response to changing load on different nodes (**load balancing**).

In Parlog, work corresponds to processes, so dynamic reallocation of work corresponds to process migration: the movement of processes from one node to another. Process migration is normally a complex process [Powell and Miller, 1983]. However, as the state of a Parlog process consists of references to Parlog terms, and distributed unification provides Parlog with global reference, the migration of Parlog processes is straightforward.

This section shows that process migration can be programmed in Parlog without the assistance of kernel mechanisms. It first describes a methodology due to Taylor *et al.* [1987a], which permits process mapping to be programmed in a parallel logic language. A scheme that permits load balancing mechanisms to be programmed in Parlog is then presented. Both these schemes permit the execution of a goal to be initiated on a single node on a multiprocessor and subsequently distribute work to other nodes at run-time. It is shown that the distributed metacontrol mechanisms described in Sections 6.1 and 6.2 can be adapted to deal with tasks distributed over several nodes using these process migration techniques.

6.3.1 Taylor *et al.*'s Process Mapping Methodology

Taylor *et al.* [1987a] present an elegant methodology for process mapping based on program transformation. This permits programs to be augmented with mapping notations [Shapiro, 1984b], which are interpreted as requests to execute particular processes on other nodes. Process migration is achieved by a program transformation which translates annotated processes into messages representing process mapping requests. A transformed program generates a stream of mapping requests when executed. These requests are passed to processes executing on other nodes, which create the annotated processes. Process migration is hence implemented in the language, by message passing.

The transformation process is illustrated using the procedure rev, which contains a process annotated @fwd. rev(Xs,Ys) has the logical reading: Ys is the list Xs, reversed. Both the original, annotated procedure and the transformed procedure, which generates a process mapping request, are presented in Program 6.4. The mapping request generated is italicized. (app/3 is assumed not to generate mapping requests and is hence not transformed). The transformed program takes as an additional argument the module M containing the code for reverse. This is included in the message, so that

reverse can be executed on another node.

```
mode rev(Xs?,Ys↑), app(Xs?,Ys?,Zs↑).

rev([X | Xs], Ys) ←
    rev(Xs, Zs)@fwd,
    app(Zs, [X], Ys).
rev([ ], [ ]). rev(M, [ ], [ ], [ ]).

app([X |Xs], Ys, [X |Zs]) ← app(Xs, Ys, Zs).
app([ ], Ys, Ys).
```

(a) **Original**

```
mode rev(Module?, Xs?, Ys↑, Requests↑).

rev(M, [X | Xs] ,Ys,
        [fwd(M, rev(M,Xs,Zs,Rs), Rs)]) ←
    app(Zs, [X], Ys).
```

(b) **Transformed.**

Program 6.4 Process mapping through transformation.

At run-time, a transformed program is executed concurrently with a network of **monitor** processes. The monitor processes are connected by streams and typically each execute on a different node. The stream of mapping requests generated by the transformed program is passed to one of these monitor processes. This interprets the requests and forwards them to other monitors. These begin to execute the processes represented by the requests. Process mapping requests generated by these processes are forwarded in turn. Execution of the program can thus spread over all nodes on which monitors are located.

The number of monitors and the way in which they are connected define the topology of the **virtual machine** on which the program executes. Different topologies — such as ring, grid and torus — can be used, depending on the nature of the underlying hardware and the application. Program annotations may be adapted to a particular virtual machine topology: *fwd* for a ring; *north*, *south*, *west* and *east* for a grid or torus; etc.

Program 6.5 implements a simple monitor that defines a node in a ring virtual machine. This processes fwd(M,G,Rs) messages received on its In stream by initiating execution of a goal G using module M. The stream of process mapping requests Rs generated by this goal (specified for convenience in the message) is merged into the ring process' Out stream.

```
mode ring(Instream?, OutStream↑).

ring([fwd(M, G, Rs) | In], Out) ←        % Receive an input request
      M@G,                               % Start evaluating the process.
      merge(Rs,Out1,Out),                % Merge mapping requests.
      ring(In, Out1).                    % Iterate to process more input
ring([ ], [ ]).
```

Program 6.5 Ring monitor for process mapping.

A ring virtual machine is created by connecting several of these monitors so that each monitor consumes another monitor's output stream as its input stream. For example, a ring of three monitors can be defined by the conjunction:

```
?- ring([<request> |R0], R1), ring(R1, R2), ring(R2, R0)
```

where <request> represents a request entered in the ring to initiate execution.

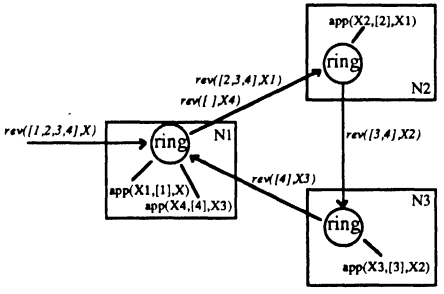


Figure 6.4 Process mapping in a ring.

Assume that a process mapping request representing a goal reverse([1,2,3,4],X) is to be executed on this ring. It is assumed that each ring process executes on a different node. Figure 6.4 represents the requests and processes created. The italicized processes (*rev*) represent process mapping requests; other processes (app) represent processes created by the monitors in response to these requests.

The goal rev([1,2,3,4],X) is reduced on goal N1. This generates a request *rev([2,3,4],X1)* to node N2 and creates a process app(X1,[1],X) on N1. As

computation proceeds, further *rev* messages are generated; these eventually wrap around the ring, as *rev([4],X3)* is received at node N1 and *rev([ ],X4)* is sent to node N2.Advantages of this methodology include its flexibility — a range of different topologies and algorithms can be implemented quite easily — and the fact that it does not require additional kernel support.

6.3.2 Load-Balancing

Taylor's process mapping methodology requires that a programmer or compiler specify how processes are to migrate at run-time. This approach can work well when the programmer has a good understanding of an algorithm's structure and behaviour or when communication costs are low [Taylor, 1987]. In other circumstances, studies show that load balancing schemes, which redistribute processes in response to changing load, can give better performance [Sato *et al.*,1987a]. This is because load balancing schemes only migrate processes when load becomes unbalanced. In consequence, they tend to migrate fewer processes: this can reduce the number of references to remote terms and hence reduce communication. This improves performance if communication is expensive.

Parlog's uniprocessor tasks and deadlock detection permit combined load balancing/process mapping schemes to be programmed in the language. These migrate annotated processes between nodes when nodes are idle.

Consider a monitor network such as that used in Section 6.3.1. To implement load balancing, monitors are modified to (a) execute application processes within a metacall and (b) retain mapping requests within a process list rather than immediately forwarding them to other monitors. The modified monitor only distributes them when requested by other monitors. It can thus return cached mapping requests to its task if a deadlock exception indicates that the task is idle: that is, that all processes in the task have terminated or are suspended. If its process list is empty and its task is idle, it asks other monitors for processes.

Program 6.6 implements a modified ring monitor lring/2, which uses the control metacall and the ring monitor defined in Program 6.5 to implement such a load balancing scheme. Processes in the application program are assumed to be annotated @fwd as before. This annotation is now interpreted as 'candidate for migration' rather than 'to be migrated'.

Iring/2 takes as arguments streams In and Out. These are presumed to be connected to other Iring processes in a ring. A call to Iring/2 initiates execution of a ring monitor (Program 6.5) in a metacall and creates a monitor Iring/6 (C1). Iring/6 takes as arguments streams defining a ring (In and Out), streams to and from the ring monitor inside the metacall (To and From), the metacall's status stream S and an initially empty process list.

```
mode  Iring(In?, Out↑),  Iring(In?, To↑, Status?, From?, Out↑, Processes?),  wait(Work?, To↑).

Iring(In, Out) ← call(ring(To, Fr), S, C),  Iring(In, To, S, Fr, Out, [ ]).                    (C1)

Iring(In, To, S, [P |Fr], Out, Ps) ← Iring(In, To, S, Fr, Out, [P |Ps ]). % Mapping request. (C2)
Iring(In, [P |To], [deadlock(_) |S], Fr, Out, [P |Ps]) ← Iring(In, To, S, Fr, Out, Ps).      (C3)
Iring(In, To, [deadlock(_) |S], Fr, [W |Out], [ ]) ←        % Task inactive: issue request.  (C4)
                    wait(W,To1),  merge(To1, To2, To),  Iring(In, To2, S, Fr, Out, [ ]).
Iring([W |In], To, S, Fr, Out, [P |Ps ]) ← W = P,  Iring(In, To, S, Fr, Out, Ps).            (C5)
Iring([W |In], To, S, Fr, [W |Out], [ ]) ← Iring(In, To, S, Fr, Out, [ ]).  % Forward request. (C6)

wait(W, [W]) ← data(W) : true.      % Wait for request to be granted.                         (C7)
```

Program 6.6 Ring monitor for load balancing.

Mapping requests generated by the application are intercepted by Iring and recorded in the process list (C2). If deadlock is detected in the application, processes from this list are passed to the application (C3). If deadlock is detected and the list is empty, a request variable is generated on the ring (C4). A wait process is created that waits for this request to be granted and then passes it to the application (C7). A monitor receiving a request variable on the ring instantiates it to a process, thus migrating that process, if its process list is not empty (C5); otherwise if forwards it (C6). Requests are thus generated when nodes are idle and circulate around the ring until they are serviced. No requests are generated, and no migration occurs, when nodes are busy.

To simplify presentation, Program 6.6 does not deal with status messages other than deadlock(_).

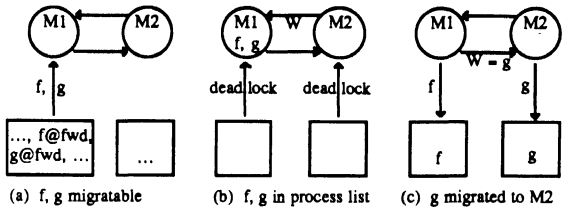


Figure 6.5 Load-balancing in Parlog.

Figure 6.5 illustrates the execution of this program on a two-node ring. M1 and M2 are lring monitors, created by a query:

```
?- lring([<request> |L], R), lring(R, L).
```

Each monitor creates and monitors a task. In (a), two annotated goals — f and g — are encountered in M1's task. Monitor M1 places these in its process list. In (b), both tasks become idle. M2 requests M1 for work and in (c), is given g, which it starts executing. Meanwhile, M1 starts executing f.

6.3.3 Distributed Metacontrol and Process Migration

Process migration appears to make the implementation of metacontrol functions such as task abortion and termination detection more difficult, as processes may exist not only on nodes but also in transit between nodes. However, if process migration is programmed in Parlog as described here, the distributed metacontrol mechanisms described in Sections 6.1 and 6.2 can be applied with little modification.

The monitor processes created to perform process migration within an application task, and the application processes that they monitor, can be executed as a single task. Monitoring and controlling this task monitors and controls both the application and the additional processes created to perform process migration. For example, recall the procedure dcall/4 (Program 6.1), which initiates controlled, distributed execution of a list of goals. A call:

```
dcall(M, [ring([R |R0],R1), ring(R1,R2), ring(R2,R0)], S, C).
```

(where M contains the code defining ring/2: Program 6.5) initiates execution of a request R on a ring of three ring monitors. Execution of the query represented by this request may be monitored and controlled using the status and control streams S and C.

The only metacontrol function not immediately available when tasks use process migration is termination detection. Monitor processes connected in a cyclic structure such as a ring will deadlock, waiting for requests from neighbours, rather than terminate when the application program they are monitoring terminates. Recall however that the deadlock exception message has a process count associated with it. Observe that when the application program terminates, tasks at each node will be deadlocked *and will have a process count of one*: the monitor. (Assuming that the monitor consists of a single process, as is the case with the monitor presented in Section 6.3.1). A simple extension to the deadlock detection algorithm described in Section 6.2 permits this situation to be detected. It can then be reported as termination. In the example, a status message [deadlock(3)|_] denotes termination of the application task; deadlock(*N*), $N > 3$, denotes deadlock.

6.4 A Multiprocessor Operating System

Figure 6.6 represents four nodes, labelled n1–n4, of a multiprocessor version of the uniprocessor OS described in Section 4.7. This OS is implemented by a program quite similar to Program 4.8. Principal differences are:

- services are distributed over the different nodes.
- the name server is duplicated at each node.
- a set of identical **mapping servers** (named node1, node2, node3, node4, etc.) are located one per node.
- there is a new **distributor** service (dist).

As before, a single user-level task is created initially. Both the task supervisor that controls this initial task and the mapping servers have stream connections to the name servers. Each name server can access every service. For simplicity, stream connections between name servers and services, the filter processes used to validate requests and the merge processes used to multiplex request streams (all illustrated in Figure 4.10) are not shown here.

This simple OS is used to motivate and illustrate a discussion of issues in Parlog multiprocessor OS design.

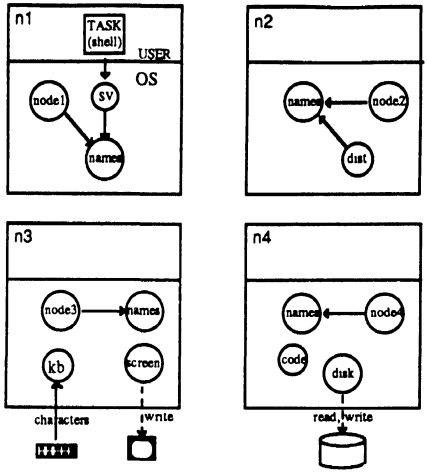


Figure 6.6 Multiprocessor operating system.

6.4.1 Replicating and Duplicating Services

A uniprocessor OS such as Program 4.8 is unlikely to perform well if ported directly to a multiprocessor. Two possible sources of inefficiency are bottlenecks, due to too frequent accesses to centralized structures, and excessive communication. To avoid these problems, it may be necessary to *distribute* and/or *replicate* services. For example, the OS illustrated in Figure 6.6 replicates the name server at each node. This avoids the name server becoming a bottleneck. It also reduces communication, as services located on the same node as a task can be accessed without inter-node communication.

The code service (code) is a candidate for distribution. Recall that the code service (Section 4.3.1) permits user-level tasks to request code modules by name. This centralized service results in potentially large terms being copied between nodes each time a program is executed. A solution to this problem is to provide a code cache at each node to cache modules retrieved from the code service. This effectively distributes the functionality of the code service. The code service described in Section 4.3.1 permits the {name,module} mapping maintained by the code service to be modified at run-time. Some mechanism is required to keep code caches consistent in the face of such modifications.

6.4.2 Physical Services

Recall that physical services are those that use side-effecting primitives (Section 4.3). It may be assumed that calls to such primitives are compiled to instructions that directly access hardware. Physical services must therefore be positioned on the nodes on which the devices they control are located. This is illustrated in Figure 6.6: the screen and disk services are located on the node on which the terminal and disk drive are located.

6.4.3 Bootstrapping and Process Mapping

In order to bootstrap the multiprocessor OS illustrated in Figure 6.6, it must be possible to create processes on particular nodes. This can easily be achieved if it is assumed that an initialization procedure creates a single mapping server process on each node plus a central bootstrap process, provided with streams to the mapping servers. Taylor *et al.* [1987a] describe how these initial stream connections can be established. In Figure 6.6, mapping servers node1, node2, node3 and node4 plus a further bootstrap process (not shown) are assumed to be created at initialization.

Program 6.7 implements a mapping server. This takes as arguments a stream of requests and an output stream to a name server. It processes requests to create processes (C1) and tasks (C2).

```
mode mserver(Requests?, NameServer↑).

mserver([process(M, G) |Rs], Ns) ← M@G, mserver(Rs, Ns).   % Create process.   (C1)
mserver([task(M, G, L, R) |Rs], Ns) ←                                            (C2)
    call(M, G, S, C),                  % Create uniprocessor task,
    lsv(S, C, L, R, Ns1),              %   and a local supervisor;
    merge(Ns1, Ns2, Ns),               %   merge requests to name server.
    mserver(Rs, Ns2).
```

Program 6.7 Mapping server.

Mapping servers permit process mapping to be programmed in Parlog using message passing. A message sent to a mapping server by a process located on another node causes a new process or task to be created on the node on which the mapping server is located. In the example, the bootstrap process can send process messages to

the various mapping servers to request the creation of the various OS processes. Shared variables in these messages implement the operating system's communication streams.

The purpose of the mapping server's task message is made clear in Section 6.5.

6.5 Multiprocessor Scheduling

The problem of allocating processor resources on a multiprocessor (the *global* scheduling problem [Wang and Morris, 1985]) can be divided into two subproblems:

- allocating nodes to tasks so as to meet goals of fair allocation of processor resources; maximum utilization; etc.
- migrating processes between nodes allocated to a task, so as to maximize throughput.

Implicit in this division is the assumption that tasks will in general be allocated disjoint sets of nodes. If they are not, a third problem, that of scheduling tasks executing on the same node (*local* scheduling), arises. It has already been shown that this can be programmed in Parlog (Section 5.4).

In Section 6.3, it was shown that process migration schemes based on both programmed process mapping and load balancing can be implemented in Parlog. This section shows how the allocation of nodes to tasks can be programmed in Parlog and integrated into a multiprocessor OS, such as that illustrated in Figure 6.6.

Mapping servers (Program 6.7), shown in Section 6.4.3 to permit the creation of processes on particular nodes, can also be used to create tasks. Clause C2 of this program processes a message task(M,G,L,R) by creating a local supervisor and a new uniprocessor task to evaluate goal G using module M. L and R are the left and right hand sides of a circuit. These can be used to link the local supervisor with the local supervisors of other tasks. It is thus possible, by sending task messages to several mapping servers, to create the process network necessary to monitor and control a distributed task (illustrated in Figure 6.2).

The mapping of distributed tasks to the nodes of a multiprocessor can hence be programmed in Parlog without additional kernel support. The allocation of nodes to tasks (that is, the decision regarding which nodes task messages are to be sent to) can conveniently be handled by a single process (or network of processes) which processes

requests to create distributed tasks. This is the function of the distributor process (dist) illustrated in Figure 6.6. Requests made to the distributor by other processes specify a query to be executed and may include hints about what processor resources are required (such as the number of nodes, their spatial distribution, etc.).

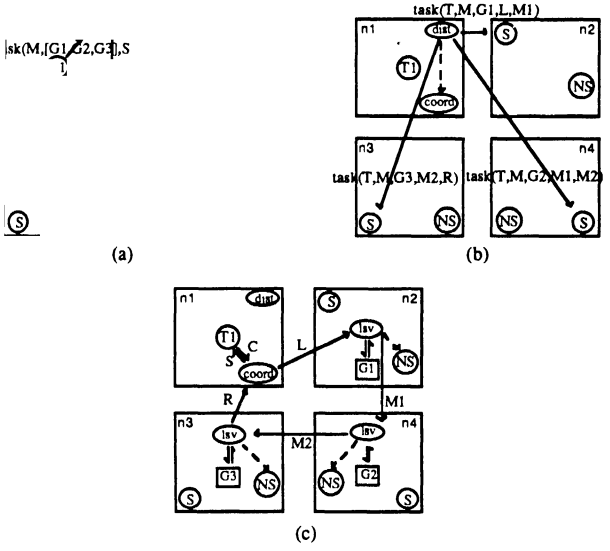


Figure 6.7 Task distribution.

Figure 6.7 illustrates the execution of the distributor. In (a), it receives a request task(M,[G1,G2,G3],S,C) which it interprets as a request to initiate distributed execution of a conjunction of goals (G1,G2,G3). In (b), it determines that it can allocate three nodes to the task — n2, n3 and n4 — and generates messages to mapping servers (S) on these nodes: task(M,G1,L,M1), task(M,G1,M1,M2), task(M,G1,M2,R). It also creates a coordinator: coord(L,R,S,C). In (c), the task messages are received and processed by the mapping servers. A distributed process network equivalent to that illustrated in Figure 6.2 is created. The task that requested the distributor to create the distribute task can now monitor and control its execution using status and control streams S and C.

This shows how the distributor deals with statically partitioned tasks. The distributor can also deal with tasks that are to be partitioned dynamically using the process migration techniques described in Section 6.3; it simply requests mapping servers to create the monitor processes that implement process migration.

To provide application programs with the ability to create distributed tasks, new system calls are defined. For example, a call dtask(M,Gs,S,C) requests distributed execution of a conjunction of goals Gs. This system call is implemented in the same way as the uniprocessor system call task/4 (Program 4.10: C2), except that the new task is created by sending a message to the distributor rather than by the use of the control metacall.

Note that the mapping server provides each local supervisor that it creates with a stream to a name server located on its node (Program 6.7: C2). As the dotted lines in Figure 6.7 (c) illustrate, this permits an extended local supervisor to route exceptions representing send system calls (Section 4.4) directly to a local name server (NS) instead of forwarding them to the coordinator. This tends to reduce communication. This optimization illustrates the flexibility of this approach to metacontrol; as distributed metacontrol functions are programmed in the language, a range of different types of distributed metacalls can be implemented.

202

CHAPTER 7.

A Language for Metaprogramming

This chapter describes the design and implementation of the parallel logic programming language Parlog+. This language is more expressive than the Parlog language from which it is derived in the sense that it permits more succinct and elegant solutions to certain metaprogramming and systems programming problems.

The main objective in the design of the language is to provide a declarative treatment of program update in a parallel logic language. Section 7.1 describes why this is desirable and indicates problems that must be overcome to achieve this. It also proposes solutions to these problems.

Section 7.2 defines Parlog+. This language extends Parlog with three principal abstractions: a state data type, persistence and atomic transactions. The application of the language is illustrated with examples. A comparison of Parlog+ with other attempts to provide a declarative treatment of state change follows.

The primary role envisaged for Parlog+ is as a programming environment and applications programming language in a Parlog operating system. A Parlog implementation of Parlog+ provides the programmer with metaprogramming primitives, a persistent file system and a declarative semantics for program update. Section 7.4 describes an approach to the implementation of principal Parlog+ language features in Parlog.

7.1 The Problem of State Change

Much of the power of declarative languages stems from the fact that a programmer only needs to specify relations between objects. He need not specify how members of these relations are to be computed. As a result, declarative programs can be more succinct, elegant and easily understood than equivalent imperative programs [Backus, 1978; Kowalski, 1983].

The declarative language programmer spends much of his time analyzing, transforming, testing and managing programs. Tools used to perform such tasks should thus be easy to write, modify and understand. Ideally, they should be

declarative in nature so that they can be analysed, transformed, etc. in the same way as the programs they manipulate. However, program development and management tasks generally involve modifying the contents of secondary or *stable* storage (referred to here, without wishing to imply any particular organization, as the **file system**). Most declarative languages rely on non-declarative, side-effecting primitives to achieve such modifications. Declarative language programs that modify the file system must therefore specify explicitly the sequence of actions to be performed to move from one file system state to another. That is, they must be written in an imperative style.

The disadvantages of an imperative programming style have been argued eloquently elsewhere. Apart from difficulties in comprehensibility, programs with imperative features cannot readily be *executed in parallel.* For example, consider a Parlog procedure modify that reads, transforms and then updates a program at a specified address using side-effecting primitives read and write:

```
modify(Addr) ← read(Addr,Pr) & transform(Pr, NewPr) & write(Addr,NewPr).
```

Two concurrent calls to this procedure:

```
?- modify(1), modify(1).
```

may transform the program at address 1 once or twice, depending on the temporal ordering of the read and write operations.

Second, imperative primitives make *logical failure* problematic, as a file system can be left in inconsistent states. For example, if a call to modify fails, the file system may still have been modified.

The two problems noted here can be solved in an imperative formalism by providing linguistic support for mutual exclusion and atomic actions [Hoare, 1974; Weihl and Liskov, 1985]. However, this makes already complex programs even harder to understand. A purely declarative treatment appears preferable.

A declarative treatment of file system update is described in this chapter. This permits the specification of file system updates as relations between states of the file system. For example, it enables modify to be expressed as a relation: modify(A,S,S1) with the logical reading: S is a file system state, and S1 is the file system state that results when program A in state S is transformed.

As state is represented by a language term, calls to modify can be composed while preserving declarative semantics. For example, the conjunction:

```
... , modify(A, S, S1), modify(A, S1, S2), ...
```

computes the relation: S is a state, and S2 is the state that results when program A is transformed *twice*.

As will be seen, file system update is viewed as a consequence of the computation of members of relations defined by programs such as modify. Logical failure is hence not a problem: if a query fails to compute a member of a relation, no update occurs.

In the following discussion, the term 'state of the file system' will generally be abbreviated as simply 'state'.

7.1.1 Problems

The following problems must be overcome to achieve a declarative treatment of state change in a parallel declarative language.

Representing State and State Change

A declarative treatment of change requires a declarative formalism in which the state of the file system is accessible in the language. It then becomes possible to write programs (such as modify) that describe relations between states and which can be executed to compute new states.

This raises the problem of how to represent state in the language. The state of the file system may be made accessible in the language as strings of bytes. This is the most general representation but also the least useful. As the declarative language programmer is primarily concerned with developing, testing, etc., declarative language programs, it may be more useful to support a more abstract view of file system state as a set of such programs.

The representation chosen must permit an efficient and succinct representation of change. A programmer is generally concerned with making quite minor changes to the file system. It should thus not be difficult to construct a new state which differs only in some minor respect from a previous state. The problem of efficiently representing the fact that some state differs from another in a single component (without using destructive assignment) has been termed the frame problem [Raphael, 1971].

Implementation of State Change

A program called with a representation of file system state as an argument can compute a representation of a modified state. For example, if S represents a file system state, a call `modify(A,S,S1)` may compute a new state `S1`. The language implementation must translate this computation of a new state (S1) to physical changes

to the file system it represents. This includes the problem of efficiently determining how new representations differ from old.

Self-Reference

If a programming system is to be self-contained, programs must be able to access and update their own representations in the file system. A call modify(A,S,S1) should be able to compute a new state in which it is differently defined.

Concurrency

Any declarative treatment of state change to be integrated into a parallel language such as Parlog should not prevent parallel evaluation. This suggests a need for concurrent access to and update of state components. Concurrency raises semantic and implementation problems. To preserve declarative semantics (which assumes that a query is evaluated with respect to an unchanging state) it should not be possible to update a state component whilst it is being accessed. This implies a need for mutual exclusion mechanisms in an implementation, and consequent problems of deadlock, etc. Some of these problems are related to problems encountered in distributed databases [Bernstein and Goodman, 1981; Ullman, 1983].

7.1.2 Solutions

The design of the parallel logic programming language Parlog+ applies the following solutions to these problems. Some of these concepts are well-known in metaprogramming, databases and programming language design. However, their integration in a logic programming language is new.

State as Language Term

A new **state** data type is introduced into a declarative language. A state is a representation of a file system Programs can thus access representations of state components and compute representations of new states.

Components of a state can be executed or accessed without concern for their *location* in primary or secondary storage. Furthermore, as noted below, representations of new states can be computed without concern for their *longevity* after computation terminates. Atkinson *et al.* [1986] refer to this incorporation of the file system in the language as **persistence**. Persistence removes the distinction that most languages make between accessible but volatile primary storage and less accessible but stable secondary storage. Programs in a persistent language simply access *language*

objects; these are loaded and saved by the language implementation, as required. As it has been suggested that typically 30% of programs is concerned with moving data to and from disk and converting formats used for disk storage to language structures [Atkinson *et al.*, 1986], this is a significant advantage.

An implementation of Parlog+ must thus translate references to state components into commands required to load and save data on secondary storage.

State as Logic Program

The 'state' made accessible to programs is presented as a set of terms representing *logic programs* rather than files, records or strings of characters. This abstraction facilitates metaprogramming: the writing of programs that analyse and transform other programs [Bowen and Kowalski, 1982].

Attributes

Conventional file systems normally associate, implicitly or explicitly, data such as creation date, format, etc. with files. A representation of state as a set of logic programs makes the specification of this 'metainformation' difficult. It can only be represented as logic clauses, located either in the program to which it refers or in some other program. The former option confuses programs and metainformation; the latter leads to the problem of relating metainformation to the program to which it refers. These problems are avoided in Parlog+ by permitting labelled terms named **attributes** to be associated with state components. An attribute is a {name,value} pair. Thus a program may have an attribute {created_by,john}, a procedure an attribute {comment,'A procedure to sort lists'}, etc.

Concise Representation of State

The state data type represents a concise 'encoded representation' of state. Language primitives are provided that can be applied to these encoded representations to access particular components. Other primitives are provided that permit the description of new states in terms of how they differ from previous states. This *avoids frame problems*: it is easy to construct new states that only differ from another state in small details. It enables an implementation of Parlog+ to represent new states efficiently; only the modifications required to construct the new state need to be recorded. Finally, it makes it easy to determine how states differ. An implementation can readily determine the physical changes to the file system that are implied by a new state.

Separation of Computation and Update

Updates to a file system are necessary. However, the side-effects required to effect update need not pervade a programming language's operational semantics. Instead, they may be defined to be a consequence of the declarative reading of programs. This is achieved as follows. Certain programs are interpreted as defining relations between states of the file system. Such a program can be executed with a representation of the current state as input to compute a new state as output. The original state — which contains the program with respect to which the query was executed — is then replaced with the output state. Further queries are evaluated with respect to the new state.

This approach, proposed by Backus [1978] in the context of functional programming, provides update without compromising the declarative semantics of programs that define relations between states. It permits self-reference: a program can access its own representation in the current state and can compute a new state in which it is differently defined. Programs describing state change can be composed to build more powerful programs. Furthermore, failure does not lead to inconsistent states: update only occurs following successful termination of a query.

Nested Transactions and Concurrency Control

The evaluation of a query, and the replacement of the state with respect to which it was evaluated with a new state that it has computed, is termed a **transaction**. Like transactions in database systems [Ullman, 1983], it represents an update to be performed either in its entirety or not at all.

The ability to nest transactions can be useful in certain applications. In systems programming, for example, it permits the programming of command interpreters that evaluate a number of queries as independent transactions.

If nested transactions are supported, several transactions may execute concurrently. If each concurrent transaction is to retain its declarative semantics, concurrency control mechanisms [Bernstein and Goodman, 1981] are necessary to avoid conflicting accesses and updates to the same state components by concurrently executing transactions. These should ensure that concurrent transactions are **serializable**: that is, that for any initial state and set of successfully terminating concurrent transactions, there exists a sequential ordering of their execution that gives the same final state. Each concurrent transaction then executes as if no other transactions are executing. A concurrency control algorithm may need to abort transactions if this condition would be violated. If any updates computed by a transaction are applied as an atomic action (that

is, *all or nothing*), aborted transactions can be restarted.

Nested transactions may be implemented as true subtransactions [Mueller *et al.*, 1983], in which case commitment of an enclosed transaction must be undone if the enclosing (parent) transaction subsequently fails. Alternatively, nested transactions may commit independently of their parent. The latter approach is adopted here, for two reasons. First, it is easier to implement. Second, it appears more appropriate for systems programming applications. For example, it was considered that failure of a command interpreter should not cause updates computed by queries that it has executed as subtransactions to be undone.

7.2 Parlog+: an Extended Parlog Language

Parlog+ is a parallel logic programming language derived from Parlog that incorporates the concepts introduced in the previous section. Section 7.2.1 outlines the basic principles of the language. Sections 7.2.2–7.2.4 describe important Parlog+ primitives and illustrate, by means of simple examples, applications of the language.

Programs presented in this section are italicized to emphasize that they are Parlog+ rather than Parlog programs. Calls to Parlog+ primitives are in boldface.

7.2.1 Overview of Parlog+

1. A Parlog+ **state** consists of a number of programs, each with a unique name.
2. A Parlog+ **program** consists of zero or more procedures and zero or more attributes.
3. A Parlog+ **procedure** is a Parlog procedure (Section 3.1) augmented with zero or more attributes.
4. A Parlog+ **attribute** is a pair {T1,T2}, where T1 and T2 are terms (Section 3.1). T1 is the **name** and T2 the **value** of the attribute.
5. Computation in Parlog+ is initiated by a **transaction** which maps a current state to a next state. A transaction specifies a conjunction of goals and the name of a program in the current state, with respect to which these goals are to be executed. A Parlog+ transaction is evaluated in the same way as a Parlog query, except that Parlog+ primitives may be called and commitment (defined below) must be performed upon successful termination.

6. A Parlog+ transaction can use metalogical primitives to:
   - execute procedures located in other named programs.
   - obtain a representation of the state with respect to which it is being evaluated (the **current** state).
   - construct representations of new states that differ from the current or other states in specified ways.
   - access representations of state components: procedures, attributes, etc. (These are represented as Parlog terms).
   - nominate a state to replace the current state upon successful termination of the transaction (the **next** state).
7. If a Parlog+ transaction terminates successfully, and has nominated a *next* state, then **commitment** attempts to replace the current state with this new state. This replacement is performed as an atomic action and either succeeds or is aborted.
8. Parlog+ programs can use a primitive akin to Parlog"s control metacall to initiate, monitor and control **nested** transactions. Nested transactions are evaluated with respect to the state as defined at the time they are initiated. They commit independently of their parent transaction.
9. A **concurrency control** mechanism ensures that all committing transactions are serializable. It may abort a transaction if committing it would violate serializability.
10. The concurrency control mechanism used in an existing Parlog implementation of Parlog+, and described herein, applies this constraint on updates:

> *A next state computed by a successfully terminating transaction may be committed* if *it does not modify any program which a concurrently executing (that is, not yet terminated) transaction has already executed or accessed using Parlog+ primitives.*

Parlog+ extends Parlog's syntax and operational semantics in three respects, as illustrated in Figure 7.1.

State and State Change

Parlog+ extends Parlog with a new data type: the state. A state is a set of Parlog+ programs. Metalogical primitives can be applied to a state to access representations of or to execute its components. Other primitives can be applied to states to generate new states. Parlog+'s metalogical primitives support self-reference: programs can refer to and modify their own representation.

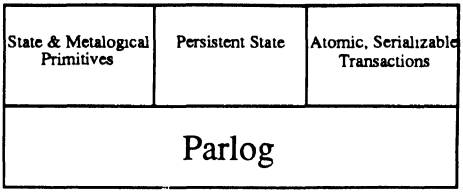


Figure 7.1 Parlog and Parlog+.

A state can be regarded as the representation of a file system: each Parlog+ program is a distinct file. (To simplify presentation, the set of programs is assumed here not to be structured in any way, so the file system is *flat*). Parlog+ programs can also be used to structure larger programs. In this respect they can be regarded as modules [Parnas, 1972]. The term module is not used here, however, to avoid confusion with Parlog's 'object code module', the unit of executable object code described in Section 4.2.6.

The state abstraction is important because it permits a declarative treatment of file system update. Also, by representing state as logic programs, it facilitates metaprogramming.

Persistent State

A Parlog+ transaction can access state components using metalogical primitives without concern for their *location* in primary or secondary storage. It may nominate some new state component that it has generated as a *next* state. Upon successful termination of the transaction, if a next state has been nominated *and* if this next state does not conflict with any concurrent transactions, it is committed and replaces the current state. The programmer does not therefore need to concern himself with the *longevity* of next states computed by transactions. In other words, state is persistent.

The persistent state abstraction is important because it conceals the existence of secondary storage from the programmer.

Serializable Transactions

A Parlog+ transaction is like a Parlog query, except that it also specifies a program with respect to which the query is to be evaluated. Transactions are written here in the form:

? *<program>* : *<query>*

For example:

? *analyse*: *current(S), transform(S, S1), next(S1).*

If a successfully terminating Parlog+ transaction (such as this example) nominates a *next* state, then its evaluation computes a mapping between states:

T: CState → NState

where CState is the current state — the state with respect to which the transaction is evaluated — and NState is the next state it computes. A transaction that fails, is aborted or does not specify a next state computes the identity mapping:

T: CState → CState

An initial transaction such as that given above can create further, nested transactions. Nested transactions may execute concurrently and, if they access and update disjoint sets of state components, may also compute next states (that is, update the file system) without contention.

Serializability guarantees that for any set E of successfully committing transactions, there exists a sequential ordering S of their executions that defines the same mapping. That is:

$\forall E = \{T_i: 0 < i \leq N\}$: CState → NState
$\exists S = \{T_1, ..., T_N\}$: T_1: CState → S_1, T_2: S_1 → S_2, .., T_N: S_{N-1} → NState

Thus each transaction in a set of concurrent transactions is evaluated with respect to some unchanging state (for transactions in E, with respect to CState, S_1, S_2, ..., S_{N-1}), and defines a simple mapping between states. Each transaction hence retains a declarative semantics as a relation defining and computing a relation between states.

Serializability is important because it permits the programmer to ignore the effects of interactions between transactions.

7.2.2 State Access

State access primitives permit a Parlog+ transaction to access representations of state components. They include ***programs***, ***dict***, ***definition*** and ***attribute***. These primitives must be applied to a term representing a state. The ***current*** primitive can be

used to determine the current state: the state with respect to which a transaction is being evaluated.

State access primitives make it possible to write metaprograms — programs that take other programs as data — in Parlog+. For example, metainterpreters and program analysers.

7.2.2.1 State Access Primitives

The annotations on arguments indicate whether an argument must be supplied at the time of call (?) or is generated by the primitive (↑).

current(*S*↑)
: *S* is the current state.

programs(*S*?, *Ps*↑)
: *Ps* is a list of the names of all programs defined in state *S*.

definition(*S*?, *P*?, *R*?, *D*↑)
: *D* is a representation of relation *R* as defined in program *P* in state *S*.

dict(*S*?, *P*?, *Rs*↑)
: *Rs* is a list of the names of all relations defined in program *P* in state *S*.

attribute(*S*?, *P*?, *A*?, *V*↑)
: *V* is the value of the attribute named *A* associated with program *P* in state *S*.

Calls to state access primitives fail if the state component they are attempting to access does not exist.

The terms returned by state access primitives are *representations* of state components. For example, ***programs*** and ***dict*** return lists of constants representing program and relation names respectively. ***definition*** returns a tuple representing a procedure. This has the form:

{<name>, <modelist>, <clauselist>}

<name> is a constant: the relation name. <modelist> is a list of constants '?' or '↑'. <clauselist> is a list of tuples representing <clause>s. A <clause> has the form:

{<head>,<guard>,<body>}

<head> is a <goal>, and <guard> and <body> are lists of <goal>s. A <goal> is a

Parlog constant or structured term. The arguments to a <goal> are <term>s. A <term> is a constant, which represents itself; a structure v(<name>), which represents a variable named <name>; a structured term tuple(<termlist>), where <termlist> is a list of <term>s representing a tuple; or a list [<term> |<term>].

Thus a procedure:

```
on([E |L], E).
on([E1 |L], E) ← E1 =/= E : on(L,E).
```

is represented as a structure:

```
on, [?,?],                          % Name and mode.
[ { on( [v('E') | v('L')], v('E') ), [ ], [ ] },   % Clause 1: head & empty guard, body.
  { on( [v('E1') | v('L')], v('E') ),               % Clause 2: head.
    [ v('E1') =/= v('E') ],         %   Guard:   E1 =/= E.
    [ on(v('L'), v('E')) ] }        %   Body:    on(L,E).
]
                }
```

(This is in fact a simplification of the actual representation, which is complicated by the need to represent Parlog's sequential clause search and conjunction operators. The form presented here assumes all operators are parallel).

```
mode defined(State?, Relation?, Answers↑), on(Name?, List?),
     defined(State, Programs?, Relation?, Answers↑), on(Name?, List?).

defined(S, R, As) ←                                                    (C1)
     programs(S, Ps), defined(S, Ps, R, As).      % Check all programs.

defined(S, [P |Ps], R, [(P,R) |As]) ←                                  (C2)
     dict(S, P, Rs), on(Rs, R) : defined(S, Ps, R, As); % R defined in program P.
defined(S, [P |Ps], R, As) ←                                           (C3)
     defined(S, Ps, R, As).                  % R not defined in P.
defined(S, [ ], R, [ ]).                     % All programs checked.     (C4)

on([E |Es], E).                              % E on list if head of list. (C5)
on([E1 |Es], E) ←                                                      (C6)
     E =/= E1 : on(Es, E).                   % E on list if on tail.
```

Program 7.1 Program analysis in Parlog+.

7.2.2.2 Example: Program Analysis

The relation *defined(S,R,Ps)* (Program 7.1) has the logical reading: *Ps* is a list of pairs *{P1,R}*, ..., *{Pn,R}* such that the relation *R* is defined in the programs *P1*, ..., *Pn* in state *S*. (A relation can be defined in more than one program). This example illustrates the use of the ***programs*** and ***dict*** primitives.

defined/3 uses the ***programs*** primitive to obtain a list of programs defined in the state *S* (C1). It then calls *defined/4*, which checks each program *P* using the ***dict*** primitive to determine whether it contains the relation *R*. If it does, it outputs a pair *{P,R}* (C2); otherwise, it does not (C3).

Recall that the syntax: ? *<prog>* : *<query>* represents a transaction that executes a query *<q>* with respect to a program *<prog>*. Assume that a Parlog+ state contains the procedures in Program 7.1 in a program named ***analyse***. Then it can be executed in the transaction:

```
? analyse · current(S), defined(S, some_relation, Ps).
```

The call to the ***current*** primitive determines the current state; that is, the current contents of the file system. The *defined* relation is then called to determine in which programs *some_relation* is defined.

7.2.3 State Generation

Parlog+ transactions can call state generation primitives such as ***new_program***, ***new_definition*** and ***new_attribute*** to construct new states that differ from other states in incorporating different definitions for particular state components.

Many new states can be computed in the course of a transaction's execution. The ***next*** primitive can be called in a transaction to nominate a state that is to replace the current state upon successful termination of the transaction in which it is called. ***next*** can be called at any time in the evaluation of a transaction. ***next*** can only be called once in the course of a transaction; second and subsequent calls to the primitive are signalled as exceptions. Commitment of a *next* state is only attempted *if* the transaction terminates successfully *and* if upon termination the term specified by the call to ***next*** is instantiated to a valid state.

State generation primitives permit program transformations to be specified in Parlog+ as relations over states. For example, a relation *transform(S,S1)* can be defined, with the logical reading: *S* is a state, and *S1* is *S* transformed. This relation

can be called in a transaction:

? *<prog>* : ***current(S)***, *transform(S,S1)*, ***next(S1)***.

which uses the ***current*** primitive to determine the current state, *S*, *transform* to compute a new state *S1*, and ***next*** to nominate this as the next state. The transaction computes the update S → S1.

7.2.3.1 State Generation Primitives

new_program(S?, P?, S1↑)

State *S1* differs from state *S* in containing a new program named *P*.

new_definition(S?, P?, D?, S1↑)

State *S1* differs from state *S* in containing definition *D* in program *P*. *D* is a representation of a Parlog procedure. The name of the procedure is implicit in this definition.

next(S?)

The state *S* is to replace the current state upon successful termination of the transaction in which this call is executed.

7.2.3.2 Example: Program Transformation

Program transformations can easily be defined in Parlog+. State access primitives are used to obtain representations of state components. State generation primitives are used to generate new states that contain transformed versions of these components.

Program 7.2 specifies a generic transformation program. *transform(T,S,S1)* has the logical reading: *S* is a state, and *S1* is *S* transformed by a transformation *T*. A transformation is specified by a tuple: *{Rs,P,A}*, which indicates that the relations named on the list *Rs* are to be transformed using the relation named *A* as defined in the program named *P*.

access(S, Rs, Ds) has the reading: *Rs* is a list of {program,relation} name pairs and *Ds* is a list of terms representing these relations as defined in state *S*. The list *Ds* is constructed by retrieving the definitions of the named relations using the state access primitive ***definition*** (C2,3).

apply(Ds,P,A,Ds1) has the reading: *Ds* is a list of procedure definitions, and *Ds1* is that list when each definition is transformed using the procedure for the relation

named *A* in program *P*. A call to *A* is constructed for each procedure on the list using the =.. primitive. This call has the form: *A(D,D1)* and reads: *D* is a procedure definition and *D1* is related to *D* by relation *A*. The newly constructed goal is evaluated in the program *P* using the **#** primitive (see Section 7.2.4 below).

```
mode  transform(Trans?, State?, State1↑),  access(State?, Relns?, Defns↑),
      apply(Defns?, Prog?, Reln?, Defns1↑),  record(State?, Relns?, Defns?, State1↑).

transform([Rs,P,A], S, S1) ←                                              (C1)
      access(S, Rs, Ds), apply(Ds, P, A, Ds1), record(S, Rs, Ds1, S1).

access(S, [(P,R) |Rs], [D |Ds])) ←                                        (C2)
      definition(S, P, R, D), access(S, Rs, Ds).   % Retrieve current definition.
access(S, [ ], [ ]).                                                      (C3)

apply([D |Ds], P, A, [D1 |Ds1]) ←                                         (C4)
      G =.. [A, D, D1], P#G, apply(Ds, P, A, Ds1).   % Construct call to A(D,D1);
apply([ ], P, A, [ ]).                               % call this to compute D1   (C5)

record(S, [(P,R) |Rs], [D |Ds], S2)) ←                                    (C6)
      new_definition(S, P, D,S1), record(S1, Rs, Ds,S2).   % New defn in new state.
record(S, [ ], [ ], S).                                                   (C7)
```

Program 7.2 A generic program transformation in Parlog+.

record(S,Rs,Ds,S1) has the reading: *S* is a state, and *S1* is the state obtained from *S* by redefining the relations on the list *Rs* with the definitions in the list *Ds*. The state generation primitive ***new_definition*** is used to construct the new state.

Assume that the procedures presented in Program 7.2 are located in a program named *analyse*. Then *transform* can be executed in the transaction:

```
? analyse : current(S), transform(<t>, S, S1), next(S1).
```

where <*t*> represents a transformation (an example of a possible transformation is given below). This transaction computes the relation *transform(<t>,S,S1)*, where *S* is the current state, and applies the state change $S \rightarrow S1$ upon successful termination.

```
mode  abort(Proc?, AProc↑),  abort_cls(Cls?, ACls↑),  extra_arg(Goal?, AGoal↑),
      abort_gs(Gs?, AGs↑),  args(Mode?, Args↑),  primitive(Goal?).

abort([Name, Mode, Cls], [Name, [? |Mode], [[NewG, [ ], [ ]] |ACls]] ) ←          (C1)
      NewG =.. [Name, stop) |Args],          % Construct head of new clause.
      args(Mode, Args),                      % Construct args of new head.
      abort_cls(Cls, ACls).                  % Transform other clauses.

abort_cls([ {H, G, B} |Cls], [[AH, [var(v('_1')) |AG], AB} |ACls] ) ←            (C2)
      extra_arg(H, AH),                      % Transform clause head.
      abort_gs(G, AG),                       % Transform guard calls.
      abort_gs(B, AB),                       % Transform body calls.
      abort_cls(Cls, ACls).                  % Transform other clauses.
abort_cls([ ], [ ]).                                                            (C3)

abort_gs([Goal |Gs], [Goal |AGs]) ←                                             (C4)
      primitive(Goal) : abort_gs(Gs, AGs);
abort_gs([Goal |Gs], [AGoal |AGs]) ←                                            (C5)
      extra_arg(Goal, AGoal), abort_gs(Gs, AGs).

extra_arg(Goal, AGoal) ←                                                        (C6)
      Goal =.. [Name |Args],  AGoal =.. [Name, v('_1') |Args].

args([ ], [ ]).                          % Produce list of variables ('_').     (C7)
args([_ |Mode], [v('_') |Args]) ← args(Mode, Args).                             (C8)
```

Program 7.3 Program transformation in Parlog+.

The relation *abort* in Program 7.3 specifies a transformation that can be applied using the generic transformation procedure *transform* (Program 7.2). This makes a procedure 'abortable' by adding an extra argument which, when instantiated to a constant *stop*, causes evaluation of the procedure to terminate.

To illustrate the transformation, the procedure *on/2* is shown transformed and untransformed:

```
mode on(?, ?).
on ([E |L], E).
on([E1 |L], E) ←
      E =/= E1  :  on(L, E).
```

(a) Untransformed.

```
mode on(?, ?, ?).                                    (a)
on (stop, _, _).                                     (b)
on (_1, [E |L], E) ← var(_1) : true.                 (c), (d)
on (_1, [E1 |L], E) ← var(_1), E =/= E1 :            (c), (d)
      on (_1, L, E).                                 (c)
```

(b) Transformed.

The annotations in this program, (a), (b), etc. refer to the following list of transformations specified by Program 7.3:

(a) add an extra input argument to a procedure's mode declaration, '?' (C1)

(b) add an extra clause to the procedure: on(stop,_,_) (C1)

(c) add an extra argument — the variable _1, represented in Program 7.3 as v('_1') — to the head of each clause (C2) and to each goal in each clause's guard and body which is not a primitive (C4-6).

(d) add an extra guard test var(_1) to each clause (C2)

Recall that state access primitives provide Parlog+ programs with *representations* of procedures, etc. *abort* specifies a relation over representations of procedures. If it is called with the term representing the procedure *on/2* (Section 7.2.2.1) as input:

```
{
  on, [?,?],                                  % Name and mode.
  [ { on( [v('E') | v('L')], v('E') ), [ ], [ ] },   % Clause 1: head & empty guard, body.
    { on( [v('E1') | v('L')], v('E') ),       % Clause 2: head.
      [ v('E1') =/= v('E') ],                 %   Guard.
      [ on(v('L'), v('E')) ] }                %   Body.
  ]
}
```

it generates as output:

```
{
  on, [?, ?,?],                                                  %                  (a)
  [ { on( stop, v('_'), v('_') ), [ ], [ ] },                    % Clause 1.        (b)
    { on( v('_1'), [v('E') | v('L')], v('E') ), [ var( v('_1') ) ], [ ] },   % Clause 2.   (c), (d)
    { on( v('_1'), [v('E1') | v('L')], v('E') ),                 % Clause 3.        (c)
      [ var( v('_1') ) ], v('E1') =/= v ('E') ],                 %   Guard.         (d)
      [ on(v('_1'), v('L'), v('E')) ] }                          %   Body.          (c)
  ]
```

In this transformed representation of on/2 (which corresponds to the transformed source, on/3, given above), additional components are in boldface; annotations refer once again to the list of transformations above.

Note that in Program 7.3, *primitive(G)*reads: *G* is a call to a Parlog+ primitive.

Assume that the procedures defined in Program 7.3 are located in a program pabort. Then the transaction:

? *analyse* : ***current(S)***, *defined(S, on, Rs), transform({Rs,pabort, abort}, S, S1),* ***next(S1)***.

(where *defined/3* is defined in Program 7.1 and *transform/3* in Program 7.2) uses the relation *abort/2* to transform all definitions of a relation *on* in the current state. The call to *defined* yields a list of definitions of *on*; *transform* generates a new state in which these are transformed using the relation *abort*.

7.2.4 Transactions and Programs

The ***transaction*** primitive is used to initiate, monitor and control nested transactions. Both the ***transaction*** and the **#** primitives support calls to procedures located in other programs. (By default, a Parlog+ procedure call is evaluated in the program in which the procedure that made the call is located).

One application of the ***transaction*** primitive is the programming of command interpreters. Concurrent user queries can be executed as separate transactions and can access and update components of the same state. Parlog+'s concurrency control mechanisms prevent conflict when concurrent queries access the same state components.

The ability to call procedures located in other programs permits the use of Parlog+ programs to structure larger programs. A program that consists of logically distinct components can be divided into several subprograms. Inheritance schemes can be implemented by searching other programs for definitions for undefined relations.

7.2.4.1 The Transaction and Program Primitives

transaction(P?,G?,S↑,C?)

Initiates a transaction to execute goal *G* using program *P*. *S* and *C* can be used to monitor and control the transaction in the same way as the status and control streams of the control metacall (Sections 3.4, 4.2.2). Upon successful termination of the transaction, an attempt is made to commit a next state, if one

has been nominated (using ***next***) during its execution. This attempt may fail, due to conflict with another transaction. Commitment failure is signalled by the additional termination status *commit_error(_)*. The argument to this message indicates both why commitment failed and the updates that were not performed. If commitment succeeds, the usual termination status (*succeeded*) is returned.

P?#G?

Executes goal *G* in program *P*.

S?:P?#G?

Executes goal *G* in program *P*, as defined in state *S*.

Both the ***transaction*** and the first form of the **#** primitive refer implicitly to (and are evaluated with respect to) the current state: that is, the state with respect to which the transaction that calls them is being evaluated. The second form of the **#** primitive permits evaluation of goals using programs defined in new states generated by a transaction in the course of previous computation (but not yet committed).

Note that a conjunction *current(S), S:P#G*, is equivalent to a call *P#G*.

7.2.4.2 Example: An Inheritance Shell

A command interpreter or **shell** is a program that manages the execution of other programs. A simple shell is presented here to illustrate the use of Parlog+'s ***transaction***, **#** and ***attribute*** primitives.

This shell extends Parlog+'s standard operational semantics by permitting calls to undefined relations to be evaluated in other programs. It effectively implements a simple inheritance mechanism: each program may inherit definitions from one other program. It is assumed that this simple inheritance relation is specified by an attribute *inherits* associated with programs. The value of a program's *inherit* attribute specifies the 'auxiliary program' from which that program inherits definitions. Thus an attribute {inherits,library} associated with a program indicates that calls to relations undefined in that program should be evaluated using procedures in library.

The shell implemented in Program 7.4 executes each query *{P,G}* received on its input stream as a separate transaction using the ***transaction*** primitive (C1). The ***current***, ***attribute*** and ***dict*** primitives are used to obtain a list of relations defined in the program *P*'s auxiliary program, if program *P* has an attribute *{inherits,Aux}* associated with it, and if program *Aux* exists (C2,3).

```
mode shell(Queries?), monitor(Program?, Status?, Control↑),
     monitor(InheritedProgram?, Dictionary?, Status?, Control↑),
     name(Goal?, Name↑).

shell([{P,G} |Qs]) ←                                                    (C1)
     transaction(P, G, S, C), monitor(P,S, C), shell(Qs).

monitor(P, S, C) ←                                                      (C2)
     current(S), attribute(S, P, inherits, Aux), dict(S, Aux, Rs) :
     monitor(Aux, Rs, S, C);    % Find dict of auxiliary program, if it exists.
monitor(P, S, C) ←                                                      (C3)
     monitor(_, [ ], S,C).                  % No auxiliary prog: Dict = [ ].

monitor(Aux, Rs, [exception(undefined, G, Cont) |S], C) ←               (C4)
     name(G, N), on(Rs, N) :                % Find name of G. In Aux?
     Cont = Aux#G, monitor(Aux, Rs, S, C);  % Inherit G from Aux.
monitor(Aux, Rs, [exception(T, G, Cont) |S], stop).  % Other exceptions: abort.  (C5)
monitor(Aux, Rs, succeeded, C).                      % Termination ...          (C6)
monitor(Aux, Rs, failed, C).                         %                         (C7)
monitor(Aux, Rs, commit_error(_), C).                % Commit error.            (C8)
```

Program 7.4 Inheritance shell in Parlog+.

monitor then monitors the new transaction's status stream. It detects calls to undefined relations (signalled as exceptions with type *undefined*) and executes them in the auxiliary program, using **#**, if they are defined in that program (that is, if they are on the list: C4). *name(G,N)* reads: goal *G* invokes a relation named *N*. Otherwise, execution of the transaction is aborted (C5). Remaining clauses deal with termination (C6-8).

A range of more sophisticated shells can be programmed in Parlog+, using the exception message to detect calls to undefined relations; attributes or Parlog+ procedures to define inheritance structures, etc.; the ***dict*** primitive to locate relations; and the **#** primitive to initiate execution in other programs. Interesting possibilities include:

Unix-style 'search path': a shell can search through a list of programs (rather than a single program, as in Program 7.4) to locate procedures not present in the original program. The list of programs to search may be defined by a procedure specified in

some program assumed to define a user 'environment'.

Conditional and multiple inheritance: a shell can execute procedures in an auxiliary program to determine where to evaluate calls to undefined relations. For example, a relation *refer(G,P)* may be interpreted as: goal *G* may be evaluated in program *P*. A range of inheritance schemes can be specified in this way.

Restarting transactions: a shell can detect transactions aborted due to concurrency control (signalled by a *commit_error(_)* termination status) and reexecute them.

Directory structure: a shell may implement its own program access primitives and consult a program defining a 'directory structure' when processing them. Arbitrary directory structures can be defined in this way. (Primitives are implemented as exceptions, in the same way as the Parlog OS in Section 4.7 implements its system calls).

Query-the-user: a shell can ask the user for definitions for undefined relations, cache these definitions and use them to solve subsequent calls to the same relations. It can also add these new definitions to programs upon successful termination of the query that required them. This 'query-the-user' facility [Sergot, 1982], is useful in expert systems and other interactive programs in which the user must contribute to the solution of a problem. It can also facilitate top-down program development: the user is asked for definitions for undefined relations as they are encountered in a partially completed program.

7.2.4.3 Example: Possible Worlds

New states may be constructed solely for the purpose of exploring 'possible worlds': that is, executing queries using alternative definitions of a problem. Queries may be executed with respect to a new state either through metainterpretation or by direct execution. Consider the following procedures, which illustrate this and show the use of Parlog+'s ***#/3*** primitive:

```
mode equivalent1(State?,Transform?,Goal?), equivalent2(State?,Transform?,Goal?).

equivalent1(S, T, {P,G} ) ←
        demo(S, P, G), transform(T, S, S1), demo(S1, P, G).

equivalent2(S, T, {P,G} ) ←
        S:P#G, transform(T, S, S1), S1:P#G.
```

where *transform/3* is as defined in Program 7.2.

Both procedures define the same relation: a query *{P,G}* succeeds in both state *S*

and the state *S1* that results from applying transformation *T* to state *S*.

equivalent1 evaluates *{P,G}* using a metainterpreter, *demo*. *demo* uses state access primitives to access the definitions of procedures and simulates their execution.

equivalent2 evaluates *{P,G}* directly using the *#/3* primitive. Recall that a call *S:P#G* executes a goal *G* using the program *P* as defined in state *S*.

If *demo* is a correct implementation of Parlog+'s evaluation strategy, then *equivalent1* and *equivalent2* compute the same relation. One procedure may however be more efficient than the other; this depends on the relative efficiency of the state access and *#/3* primitives in a Parlog+ implementation.

7.2.5 Discussion

Parlog+ permits a declarative treatment of file system update in a concurrent language. This is achieved by extending the parallel logic language Parlog to provide linguistic support for program access to state, persistence and atomic transactions. Parlog+ programs that describe state change have a simple declarative reading. They can be composed and executed in parallel. The state change computed by a query is well-defined: failure is not problematic. Furthermore, declarative semantics is maintained when nested transactions execute concurrently.

By incorporating metalogical primitives in Parlog, Parlog+ supports metaprogramming in a parallel logic language. Metainterpreters, program analysers and program transformers can be programmed in the language. These programs can refer to and modify their own representations, so a Parlog+ state (that is, file system) is self-contained.

By making the state of a file system accessible in the language, Parlog+ allows many 'program management' and 'systems programming' tasks to be viewed as *metaprogramming* tasks. This makes the advantages of a declarative programming style available when building tools to aid in program development.

In other respects, Parlog+ inherits Parlog's properties as a systems and applications programming language.

Parlog+ language features serve to make Parlog+ a more expressive language than the Parlog language from which it is derived. However, they have an implementation cost: to provide atomic transactions, updates must be cached and then recorded on stable storage; to provide serializability, information about accesses to state components must be recorded. (The implementation of Parlog+ language features is described in

Section 7.4 below). As not all applications require atomic transactions, etc., Parlog+ is best viewed as a *programming environment* and *application programming language* for a Parlog programming system. An existing implementation of Parlog+, in Parlog, uses the language for this purpose [Foster, 1986]. This permits applications that can benefit from a declarative treatment of update to be programmed in Parlog+. Applications in which state change is more conveniently expressed in terms of communicating processes can be programmed in Parlog.

Parlog+ can be further developed in a number of areas. Two are noted here. First, Parlog+'s concept of state can be generalized. A Parlog+ state represents a set of persistent logic programs. The language can be generalized so as to incorporate in state both (a) non-persistent (that is, temporary) objects, and (b) objects other than logic programs. The former permits more efficient representation of information that needs to be shared between concurrent (nested) transactions but that is not required to persist after termination of the enclosing transaction. The latter permits a declarative treatment of other types of file system update: for example, operations on databases. In both cases, new types of state component are made available to Parlog+ programs by means of additional state access and state generation primitives.

Second, Parlog+ can be extended to serve as a language for programming fault-tolerant systems. Atomic commitment of updates is introduced into Parlog+ to provide a declarative treatment of state change. A transaction either successfully computes a state transformation or state is not modified. Commitment of the file system updates computed by a transaction is thus atomic in the face of logical failure (of a transaction) and concurrent, conflicting accesses. Commitment can also be made atomic in the face of physical (that is, hardware) failure. This provides a starting point for the use of Parlog+ to program robust or *fault-tolerant* systems: systems which behave correctly despite hardware failure.

7.3 Related Work

A number of systems have been proposed that provide solutions to some of the problems outlined in Section 7.1.1.

The Unix shell programming language's *pipe* feature [Richie and Thompson, 1974] permits programs that read and write streams of data to be linked (*composed*) in a pipeline. This allows existing tools to be combined, producing more complex shell

programs that also read and write streams of data. However, only linear, unidirectional composition is supported. Its utility is further reduced by the fact that the shell language is distinct from the languages used to program applications. Furthermore, the file system side-effects implicit in many Unix tools limit the extent to which they can be composed.

Several authors have described *functional shells* for Unix that support functional composition of programs [Shultis, 1983; McDonald, 1987]. These permit more general program composition — for example, several inputs can be passed to a single program — but do not avoid the problem of file system side-effects. In contrast, Parlog+'s declarative treatment of update means that programs that modify state can be composed. Parlog+ permits extremely general composition of programs: arbitrary communication networks can be defined as conjunctions of processes. Shared variables serve as communication channels and back communication provides added flexibility. Another advantage of Parlog+ is that it can be used as both a shell and an application programming language.

Backus [1978] proposes an approach to programming environment design that is quite similar to that presented here. In his Applicative State Transition systems, state is represented as a functional program. A simple shell repeatedly applies functions in this program to the program itself in order to obtain a new program for the next iteration. This approach permits any state-modifying program to be composed with another. However, Backus's proposal addresses neither the frame problem nor concurrency.

Bowen and Kowalski [1982] propose that logic programming systems be extended with a relation demo(Pr,G), which reads: the goal G can be proven using the program represented by the term Pr. It is then possible to construct terms representing different programs and to execute queries using these programs. However, as demo is a metainterpreter, this approach cannot be expected to give good performance. Bowen [1986] defines a language metaProlog in which demo is a language primitive and programs represented as language terms can be executed directly. In metaProlog, programs are constructed using primitives such as add_to(Pr1,Cl,Pr2), which reads: program Pr2 differs from program Pr1 in incorporating the clause Cl. Both these proposals permit a declarative treatment of change. However, problems of self-reference and concurrency are not addressed.

The language Argus [Weihl and Liskov, 1985] represents an imperative solution to some of the problems noted in Section 7.1. Its design aims not to provide a declarative treatment of update but to support fault tolerant distributed computing. It provides linguistic support for stable objects (which survive crashes) and nested atomic actions

(transactions) on these objects. Argus programs perform sequences of actions on objects; the language kernel synchronizes accesses and records changes to objects on stable storage when actions complete.

Parlog+ programs and transactions are quite similar to Argus objects and actions. One major point of difference between Argus and Parlog+ is that all Argus programs must pay the costs associated with robustness, whether this is required or not. In consequence, the language is not general purpose: it is designed specifically for use in applications requiring a high degree of reliability. Parlog+, on the other hand, is intended to be implemented in Parlog, which it enhances but does not replace. Declarative semantics for update (and potentially, fault tolerance) are hence available to the Parlog programmer, but only if required.

7.4 The Implementation of Parlog+

This section describes an approach to the implementation of Parlog+ in Parlog. As Parlog provides basic Parlog+ functions such as process reduction, unification and metacontrol, a Parlog implementation of Parlog+ only needs to support Parlog+'s extensions to Parlog. The presentation that follows is hence able to concentrate on the implementation of Parlog+'s original features. Namely: its state data type and metalogical primitives; its persistence; and its atomic, serializable transactions.

To achieve an implementation of Parlog+ in Parlog:

- Parlog+ programs are compiled to Parlog in such a way that calls to Parlog+ primitives generate exceptions when executed.
- Parlog+ transactions are executed as Parlog tasks.
- Additional Parlog processes are provided to supervise execution of Parlog+ transactions and to implement the various Parlog+ abstractions.

The principal additional processes provided are **transaction managers**, which supervise Parlog+ transactions and implement Parlog+'s state data type and metalogical primitives; and **program managers**, which coordinate accesses to Parlog+ state components and implement persistence. Transaction managers and program managers cooperate to implement atomic, serializable transactions. A **name server** routes messages from transaction managers to program managers.

Figure 7.2 illustrates a Parlog implementation of Parlog+, assumed here to be

executing as a user-level program in a Parlog operating system. Transaction managers (TM) are connected by streams to a name server (names), which routes requests to program managers (PM). These in turn communicate requests to a disk manager (disk), which provides an interface to an OS disk service (DISK). The function of transaction and program managers is made clear below. The functionality of the disk service is not relevant to the current discussion (see [Foster and Kusalik, 1986] for a discussion of issues in its implementation).

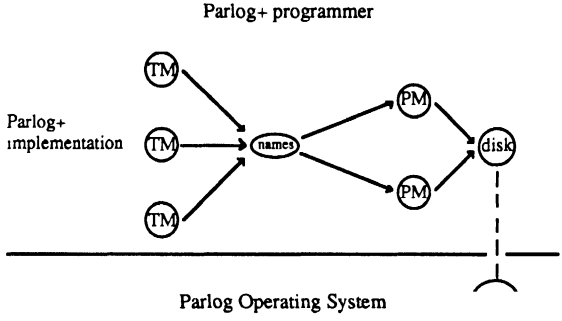


Figure 7.2 Parlog implementation of Parlog+.

7.4.1 State and State Change

Parlog+'s metalogical primitives are implemented in the same way as Parlog OS system calls: as exceptions (Section 4.4). The compilation of Parlog+ to Parlog translates calls to Parlog+ primitives into calls to Parlog's raise_exception primitive and hence at run-time into exception status messages. A transaction manager (a type of task supervisor: Section 4.4) is associated with each transaction. This monitors the transaction's status stream and traps and evaluates calls to Parlog+ primitives.

The transaction manager also implements Parlog+'s state data type. It processes calls to Parlog+'s state access and generation primitives with respect to a data structure that it maintains, termed a **virtual copy**. The virtual copy records the states generated by the transaction. Its name derives from the fact that these states are only 'virtual copies' of the current state, represented in terms of how they differ from the current state. The current state is characterized solely by a lack of modification.

Figure 7.3 represents the relationship between a Parlog+ transaction and its transaction manager.

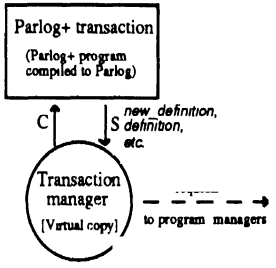


Figure 7.3 Parlog implementation of Parlog+ transaction.

Recall that the states manipulated by Parlog+ programs are 'encoded representations' of file system states (Section 7.1.2). The 'encoded representations' made available to programs are in fact indices which the transaction manager uses to access its own representation of these states, recorded in the virtual copy.

To process a call to a *state access* primitive (for example: definition(S,P,R,D)), the transaction manager must locate the state with the supplied index (S) in the virtual copy. Then, that state is searched to determine whether it contains a new definition for the required component. (In the example, the component required is the definition of relation R in program P). If it does, the new definition is used; if not, the definition of the component is determined by making a request to the manager of the program in question (see Section 7.4.2). Efficient data structures minimize the time required to locate a definition in the virtual copy.

A call to a *state generation* primitive (for example: new_definition(S,P,D,S1)) is processed by adding a representation of the newly created state to the virtual copy. The representation of a new state can be simply the representation of the state from which it was derived (state S) plus the modification used to generate it ({P,D}). S1 is instantiated to the index of the new state in the virtual copy data structure.

The transaction manager also maintains the name of any *next* state nominated by the transaction. Upon successful termination of the transaction, the transaction manager examines the virtual copy to determine whether a *next* state has been nominated. If so, a *commitment* procedure is invoked to attempt any updates that it implies. Commitment is discussed in Section 7.4.3.

7.4.2 Persistence

Parlog+ primitives enable Parlog+ transactions to execute, access representations of and modify state components without concern for their location or longevity. This *persistence* is implemented by perpetual processes termed program managers. The programs comprising state are stored on stable storage. A program manager with a request stream to disk service(s) is associated with each program. Transaction managers communicate with program managers to request Parlog terms representing source code (following calls to ***definition***, ***dict***, etc.) or executable object code (after calls to ***transaction***, ***#***). Transaction managers also request program managers to perform updates when a transaction terminates successfully having nominated a ***next*** state which modifies programs.

A program manager's principal task is to correctly service requests from transactions whilst maintaining a consistent representation of its program on stable storage. It achieves this by generating messages to disk services to store and retrieve data as required. It may also cache frequently accessed program components in memory to reduce disk traffic.

A program manager handles requests for source and object code by accessing a cache and/or communicating with disk managers to retrieve data. It handles update requests by validating the update (as described in Section 7.4.3 below), modifying its cache and communicating with disk service(s) to update the representation of the program on stable storage. Commitment of updates is discussed in detail in Section 7.4.3.

7.4.3 Atomic, Serializable Transactions

Parlog+ transactions have a simple declarative semantics. They are evaluated with respect to an unchanging current state and may compute a new state to replace the current state upon successful termination. This feature of the language requires that concurrently executing transactions are serializable, so that each transaction is evaluated with respect to an unchanging state. This in turn requires that updates computed by a transaction are applied as an atomic action, "all or nothing".

The implementation of atomic update and serializability involve both transaction managers and program managers. The transaction manager uses an algorithm called *two stage commit* to coordinate updates to ensure that they are applied atomically.

Program managers apply a *concurrency control algorithm* to verify that serializability is not violated. They also perform the actual updates.

7.4.3.1 Two Stage Commit

A form of **two stage commit** [Lampson and Sturgis, 1976] is used to commit updates computed by a Parlog+ transaction as an atomic action. In the first stage of this algorithm, a transaction manager makes a **prewrite** request to each program manager affected by an update. This requests the program manager to validate its update. Any program manager may refuse its updates at this stage; this does not matter, as no program has been modified. If all affected program managers accept their updates, then **write** requests are issued to request program managers to actually apply the updates. Updates are thus performed "all or nothing".

The Parlog implementation of two stage commit presented in this section ignores the possibility of system failure during commitment. However, if stable storage can be assumed, then extension to incorporate a recovery mechanism that ensures that commitment is also atomic in the face of hardware failure [Bernstein *et al.*, 1983] is easy.

7.4.3.2 Concurrency Control

The implementation of serializability requires concurrency control mechanisms. Concurrency control has been extensively studied by the distributed database community. The reader is referred to [Bernstein and Goodman, 1981; Papadimitriou, 1986] for a detailed discussion of this topic and for a precise definition of serializability and the concurrency control problem. Informally, concurrency control seeks to avoid or detect **conflict**. Transactions in database systems (and in Parlog+) perform a series of read and write operations to state components. A set of concurrent transactions conflict if the sequence of operations that they perform are such that the same final result cannot be achieved by executing the transactions in some sequential order.

For example, consider the following four (independent) sequence of operations. Each such **history** represents the operations that two transactions T1 and T2 were observed to perform on state components X and Y. read(T,X) signifies a read operation by transaction T on term X and write(T,X) signifies the corresponding write operation.

H1 = { read(T1,X), write(T2,X), read(T1,X) }
H2 = { write(T1,X), write(T2,Y), write(T2,X), write(T1,Y) }
H3 = { read(T1,X), write(T2,X), read(T1,Y) }
H4 = { write(T1,X), write(T1,Y), write(T2,X), write(T2,Y) }

Histories H1 and H2 both conflict, as neither the execution of T1's operations followed by T2's, nor the reverse, gives the same final result. On the other hand, neither history H3 nor history H4 conflict, as in each case the same final result can be achieved by executing first transaction T1 and then T2.

Concurrency control mechanisms either **avoid** conflict by sequencing transactions when necessary or permit transactions to execute concurrently and seek to **detect** conflict at run-time. As conflict avoidance requires prior knowledge of the behaviour of transactions, conflict detection is more commonly used.

Various conflict detection algorithms have been proposed. Those based on **locks** and **timestamps** seek to detect conflict as early as possible by requiring that transactions announce their intention to modify items. **Optimistic** approaches, on the other hand, assume that conflict is rare and therefore only check for conflict immediately prior to committing a transaction [Badal, 1979]. (Commitment is the process which makes updates computed by a transaction permanent and accessible to other transactions). All approaches resolve conflict by aborting transactions. As long as commitment of updates computed by a transaction is performed as an atomic action, an aborted transaction can be reexecuted.

An optimistic conflict detection algorithm checks for conflict immediately prior to commitment of a transaction T. It tests whether, if T is committed, the set of all committed transactions would still be serializable. In general, this requires it to keep a record of all preceding reads and writes. The overhead of maintaining this information militates against optimistic approaches. For example, assume that a transaction T2 is allowed to commit and that T2 writes a term X which an active transaction T1 has previously read. The operation write(T2,X) must be recorded, as if T1 were subsequently to commit, having read term X again, serializability would be violated: this is the conflicting history H1 above.

Nevertheless, an *optimistic* concurrency *detection* algorithm is used in the implementation of Parlog+. Conflict detection rather than conflict avoidance is used because it is not in general possible to know what programs a Parlog+ transaction will access without executing it. An optimistic algorithm is used because it cannot be known whether a next state computed by a Parlog+ transaction is to be committed until

the transaction terminates. There is thus little point in attempting to detect conflict early. The overhead associated with optimistic approaches is avoided by using what may be termed a *non-preemptive* conflict resolution algorithm: this aborts a committing transaction if subsequent action of any active transaction *could* lead to conflict. (An active transaction is one that has not yet terminated). It aborts committing transactions that would write state components other active transactions have previously read. In the example in the previous paragraph, T2 would be aborted when it attempted to commit. This algorithm only requires information on reads by active transactions, rather than reads *and* writes by *all* transactions. This is a significant saving. On the other hand, the algorithm tends to abort more transactions than other concurrency control algorithms. As these transactions may never reread these components, this abortion may be unnecessary.

In database systems, concurrency control may be applied at various levels of granularity. There is a tradeoff between the run-time cost of recording accesses to components and the convenience of concurrent updates to components. For simplicity, it is proposed that a fairly large grain-size be adopted in an implementation of Parlog+. Read and write operations are defined to occur on programs rather than particular procedures or attributes. This granularity is in fact imposed by the underlying module system in a Parlog implementation of Parlog+. This does not permit a transaction manager to determine which procedures in a program a particular transaction executes. This prevents procedure-level concurrency control.

A Parlog+ transaction is hence allowed to commit if and only if the next state it has computed does not modify programs which any other active transaction has accessed using a state access primitive or has executed.

Note that Parlog+'s concept of update as the generation of a new state that differs from a previous state in some specified way precludes the possibility of conflict *within* a transaction. Any *next* state computed by a Parlog+ transaction can only specify a single new definition for any item.

7.4.3.3 Implementation of Two Stage Commit

The commitment of a transaction's updates involves the cooperation of the transaction's manager and the managers of the programs affected by the update. The transaction manager supervises commitment. It uses two stage commit (Section 7.4.3.1) to synchronize the activities of the program managers concerned so that update of all programs occurs as an atomic action. The program managers apply concurrency

control constraints, as described in the next section. They also perform the actual updates, as noted in Section 7.4.2 above.

A Parlog implementation of two stage commit consists of commitment procedures for both the transaction and program managers. These are presented in Program 7.5.

A transaction manager invokes its commitment procedure, t_commit/3 (C1-6), when its transaction terminates successfully, having nominated a valid *next* state. t_commit is called with the transaction's termination variable, the next state as represented in the virtual copy and a stream to the name server as its arguments. It determines what updates are required and generates update messages to request the program managers concerned to perform the updates.

A program manager invokes *its* commitment procedure, p_commit/4 (C7-13), when it receives an update request. p_commit is called with a list of termination variables, before and after commitment (used for concurrency control: see the next section), a request stream to a disk manager, and the update request generated by the transaction manager as arguments.

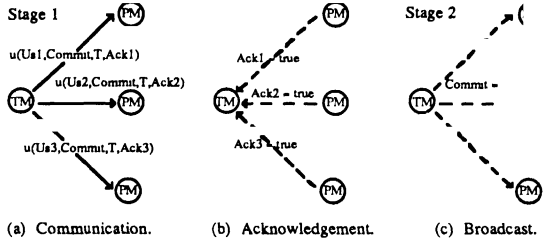


Figure 7.4 Implementation of two stage commit.

Figure 7.4 illustrates how a transaction manager (TM) and the managers of programs (PM) affected by commitment of a *next* state communicate to implement two stage commit. A transaction has terminated and nominated a next state that involves updates to three programs. In (a), the transaction manager communicates the updates (Us1,Us2,Us3) to the program managers, along with a Commit variable, unique acknowledgement variables, (Ack1,Ack2,Ack3) and the transaction's termination variable (T) (C2,3). (The relation updates(State,Ps) — not provided — reads: Ps is a list of {Program,Updates} pairs representing the program updates implied by state State). This is the *prewrite* phase of the two stage commit algorithm.

Each program manager receiving an update message calls p_commit, which validates the updates and (as described in the next section) uses inactive/2 to check that

concurrency control constraints are satisfied (C7,8). It either allows (C7) or disallows (C8) the update, and binds its acknowledgment variable to true or false respectively. valid_updates(Us), not provided, reads: Us is a valid list of updates.

```
mode t_commit (Term?, State?, Requests↑), phase2(Commit↑, Acks?, Term↑)
     phase1(Programs?, Commit?, Term?, Acks↑, Requests↑), updates(State?, Updates↑).

t_commit(T, State, Rs) ←                    % Commit updates recorded in State.   (C1)
   updates(State, Ps), phase1(Ps, Commit, T, Acks, Rs), phase2(Commit, Acks, T).

phase1([{P,Us} |Ps], Commit, T, [Ack |Acks], [{P, update(Us,Commit, T, Ack) } |Rs]) ←  (C2)
   phase1(Ps, Commit, T, Acks, Rs).     % Generate update (prewrite) requests.
phase1([ ], Commit, T, [ ], [ ]).                                                    (C3)

phase2(Commit, [true |Acks], T) ← phase2(Commit, Acks, T).     % Wait for result. (C4)
phase2(false, [Ack |_], commit_error(Ack)) ← Ack =/= true : true. % Commit error. (C5)
phase2(true, [ ], succeeded).                % All ok: signal update can proceed.  (C6)

mode p_commit(Ts?, Ts1↑, Disk↑, UpdateRequest?), inactive(Term?, Ts?).
     program2 (Disk↑, Updates?, Commit?), perform_updates(Updates?, Disk↑).

p_commit (Ts, [ ], Ds, update(Us, Commit, T, Ack)) ←  % Update request.          (C7)
   valid_updates(Us), inactive(T, Ts)   :             %   Ok to apply updates?
   Ack = true,                                        %   Signal that updates ok
   program2(Ds, Us, Commit);                          %   Yes: wait for write.
program (Ts, Ts, [ ], update(_, _, _, Ack) ) ← Ack=false.  % Not ok: signal error. (C8)

program2 (Ds, Us, true) ←                            % Wait for 'commit' signal: (C9)
   perform_updates(Us, Ds).                          %   Commit=true: proceed.
program2 ([ ], Us, false).                           %   Commit=false: abandon. (C10)

inactive(T, [T1 |Ts]) ← not( var(T1) ) : inactive(T, Ts).  % Terminated if ground. (C11)
inactive(T, [T |Ts]) ← inactive(T, Ts).                    % Ignore committing task. (C12)
inactive(T, [ ]).                                          % All terminated.       (C13)
```

Program 7.5 Transaction and program managers: committing a transaction.

In Figure 7.4 (b), all program managers allow their updates and bind their acknowledgement variables to true. Back communication occurs. In (c), the transaction manager, which has been waiting for the acknowledgement variables to be

bound (C4-6), binds the variable Commit (included in the initial messages) to true to signal that the update can proceed (C6). This informs the program managers (which have been waiting for Commit to be bound) that they can proceed to update the program (C9). This is the *write* phase of the two stage commit algorithm.

If, on the other hand, any of the acknowledgment variables is bound to false, the transaction manager binds Commit to false (C4). The program managers then discard the updates and no updates occur (C10).

In both cases, the transaction manager binds the transaction's termination variable to indicate the result of the transaction: succeeded (C6) or commit_error(_) (C5).

All procedures in Program 7.5 (except for inactive/2) have a simple logical reading. For example, t_commit(T, State, Rs) reads: Rs is the requests generated to commit a state State, and T is the result of this commitment. phase2(Commit,Acks,T) reads: *either* Acks is a list of true values, T = succeeded and Commit = true, *or* Acks is a list of values containing some element other than true, T = commit_error(_) and Commit = false. program2(Ds,Us,Ack) reads: *either* Ack = true, and applying the list of updates Us generates disk requests Ds, *or* Ack = false and no disk requests are generated.

7.4.3.4 Implementation of the Concurrency Control Algorithm

As noted in Section 7.4.3.2, concurrency control is applied at the level of programs in the Parlog implementation of Parlog+. Each program manager keeps a record of all transactions that have accessed its components. Then, when it receives an update request, it checks whether any of these transactions are still active (that is, have not terminated). If so, the update is disallowed.

A program manager is able to keep a record of all transactions that have accessed its components because a transaction manager must communicate with a program manager to process Parlog+ primitives such as ***definition*** and **#**. It is able to determine whether these transactions are still active because each such communication has a **termination variable** associated with it. Recall that a termination variable (Section 4.4.2) is a unique variable associated with a Parlog task, which the task's supervisor (in this case a transaction manager) guarantees to instantiate when the task (that is, transaction) terminates. A program manager retains references to the termination variables of transactions that have accessed its components. An update is only allowed if the termination variables associated with all previous requests (except for requests made by the committing transaction) are non-variable.

The relation inactive/2 in Program 7.5 applies the concurrency control constraint. It tests the termination variables associated with previous requests to verify that they are either instantiated, indicating termination (C12), or identical to the termination variable of the committing transaction (C13). (The latter case represents requests made by the committing transaction).

7.4.3.5 Alternative Concurrency Control Algorithms

The non-preemptive optimistic concurrency control algorithm described above prevents programs being modified whilst in use. Yet it may be desirable to be able to modify programs whilst permitting transactions already executing them to continue executing using the old version. This can be achieved without violating serializability (and hence declarative semantics), but at the cost of additional run-time overhead, using the usual optimistic concurrency control algorithms [Badal, 1979].

Alternatively, the application of concurrency control mechanisms may be made optional, on a program-by-program or alternatively a transaction-by-transaction basis. Two stage commit is still used to ensure atomicity of update, but program managers do not prevent updates if active transactions have accessed the program. Thus, in Program 7.5, the call to inactive is not required. However, this compromises serializabilty and hence declarative semantics.

7.4.3.6 Deadlock

For simplicity, Program 7.5 ignores the problem of deadlock. Upon receiving an update request from a transaction manager, a program manager invokes p_commit to execute the two-phase commit and concurrency control algorithms. The program manager processes no further messages until p_commit terminates. However, if two transactions that update the same programs attempt to commit at about the same time, the situation can arise where one of each transaction's update requests is delayed, as illustrated in Figure 7.5. update requests have been generated by two transactions, T1 and T2, to programs A and B. Program A receives T1's update message; program B receives T1's. The other update messages remain unreceived, and will remain unreceived, as neither program will accept further messages until update is completed or aborted. Yet neither transaction manager can complete or abort its update until it has received a response to *both* its messages. This is deadlock.

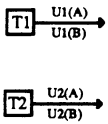


Figure 7.5 Deadlock in the implementation of two stage commit.

The solution to this problem employs a useful Parlog programming technique. A program manager attempting to commit an update must, whilst waiting for the transaction manager to indicate whether it should proceed, concurrently scan ahead on its input stream and 'abort' any pending update requests encountered by binding their acknowledgment variable to false. This ability to 'preview' pending messages, without 'receiving' them, is another example of the power of the logical variable as a communication mechanism. (Program 4.4 previews messages in a similar way).

7.4.4 Discussion

Implementation techniques for Parlog+'s three main extensions to Parlog have been described. These techniques have been used in the implementation of a Parlog programming environment which supports the Parlog+ language [Foster, 1986].

The discussion of Parlog+ implementation techniques is of interest as an illustration of Parlog's *extensibility*. Extensibility refers to the ease with which the expressive power of a language can be augmented by the definition of abstractions. As noted in Section 2.1.1, extensibility is an important attribute in a systems programming language.

In Section 4.8, three approaches to the implementation of abstractions in Parlog were identified: as procedures, as messages processed by a perpetual process and as exceptions processed by a task supervisor. The latter two approaches were observed to be more powerful as they are able to maintain a history of interactions. The implementation of Parlog+'s state and program abstractions employ these latter two approaches. Parlog+'s state data type and associated metalogical primitives are implemented by causing calls to metalogical primitives to generate *exceptions*. A task supervisor (transaction manager) interprets and processes these exceptions with respect to a data structure that represents states generated by the transaction. Parlog+'s persistent program abstraction is implemented by perpetual processes (program managers) which process *messages* that request access or modification to persistent

programs. A program manager generates further requests to an OS disk service to store or retrieve data.

The implementation of the atomic commitment and concurrency control algorithms described is also of interest because of the complexity of these distributed algorithms. Updates to a number of programs (which may be located on different nodes in a multiprocessor) are performed concurrently. A transaction manager coordinates but does not sequence these updates. The Parlog implementation of this algorithm (Program 7.5) is, it is suggested, particularly simple and concise. This is due to Parlog's support for communication and synchronization mechanisms which in other languages would have to be explicitly specified by the programmer.

Program 7.5 depends on dataflow constraints for its correct execution. However, the logical reading of procedures is shown to provide a useful check on some aspects of the program's correctness.

CHAPTER 8.

Conclusion

8.1 Parlog and Systems Programming

This monograph has presented a methodology for the design and implementation of language-based operating systems in the parallel logic programming language Parlog. This methodology treats a Parlog operating system as a layered structure. A machine language or hardware *kernel* supports the Parlog language. Services provided by Parlog *operating system* programs are used to implement programming environments and other *user-level programs*. Simple, coherent treatments of important issues in the design of each layer in this structure were provided. These were specified in sufficient detail to permit implementation of Parlog operating systems.

The kernel was defined in terms of a simple kernel language (a subset of Parlog). Uniprocessor and multiprocessor implementation schemes for this kernel language were described. A comprehensive set of techniques for operating system design and implementation permits operating systems to be implemented in the kernel language. The design of a Parlog programming environment, intended to execute as a user-level program in a Parlog operating system, was defined in terms of an extended Parlog language, Parlog+. The implementation of Parlog+ in Parlog was described.

It is argued that the following characteristics of this methodology satisfy the criteria proposed as the objectives of this research and make it a useful basis for the implementation of operating systems to support symbolic processing on multiprocessors:

Comprehensiveness. The methodology provides a coherent treatment of most major issues in operating system design.

Few extensions to Parlog. Only minor extensions to Parlog are introduced. The most significant, the metacall exception message, proves useful in several different areas. This suggests that it is a useful extension to the language.

Support for multiprocessor execution. Multiprocessor execution is supported at both the kernel and operating system levels. The kernel's distributed unification algorithms support multiprocessor execution of Parlog and its control metacall.

Operating system components presented do not require centralized structures for their implementation. For example, naming and processor scheduling schemes are programmed in Parlog and can be distributed if required.

General applicability. Operating system structures are specified in terms of just three Parlog language features: process reduction, unification and the control metacall. The operating system design and kernel implementation techniques presented herein can hence be applied to other languages with similar features.

Simple kernel. The implementation of the kernel is described in terms of extensions to an existing abstract machine. These extensions do not introduce significant additional complexity.

Efficient kernel. Experiments based on emulations of the original and extended abstract machines confirm the efficiency of the kernel design. They indicate that it is more efficient than an alternative approach to kernel design proposed by other researchers.

Functional and flexible kernel. Though simple, the kernel language is no less expressive than the Parlog language from which it is derived. It supports all functions required by a Parlog operating system, on uniprocessors and multiprocessors. It can be used to implement a range of metacontrol functions and task scheduling algorithms.

Support for user-level programming. More complex abstractions such as persistent file systems can readily be constructed using operating system services.

Declarative treatment of program update. The extended Parlog language Parlog+ permits a declarative treatment of program update in a parallel declarative language. This allows many systems programming problems to be viewed as metaprogramming problems. It permits analysis, transformation and concurrent execution of state-changing programs.

The methodology presented in this monograph demonstrates that it is *possible* to construct language-based operating systems in the high-level, declarative language Parlog. A related question not directly addressed in this research is whether it is *useful* to use Parlog as a systems programming language. As noted in Section 2.1.3, utility is in many respects a subjective notion. However, it is argued that the research reported herein justifies the following six assertions:

- Parlog implementations of distributed algorithms are in general elegant and succinct. Dynamic process and communication structures can be specified easily. (See, for example, the implementation of atomic, serializable

transactions: Program 7.5). This simplicity is due to Parlog's linguistic support for light-weight processes, concurrency, communication, synchronization, non-determinism, tasks and exceptions.

- Parlog's support for symbolic manipulation — fully recursive data structures, call-by-reference, unification, etc. — facilitates the implementation of operating systems and programming environments for symbolic languages. For example, see the implementation of Parlog+ in Parlog.
- Parlog's implicit concurrency and uniform computational model means that programs can be ported from uniprocessors to multiprocessors without significant modification. (For example, see the Parlog operating systems described in Sections 4.7 and 6.4).
- Parlog programs generally have a simple declarative reading. Although this is not a substitute for a formal specification (in, for example, Temporal Logic), it aids informal verification of their correctness. Moreover, a Parlog program is likely to be shorter than a corresponding formal specification and just as obviously correct.
- Although Parlog programs depend on dataflow constraints for their correct execution, process synchronization rarely becomes a dominant or difficult issue when writing Parlog programs. Dataflow can in general be understood in terms of simple paradigms such as stream communication, back communication, broadcast, etc.
- The Parlog language itself is simple, both in its syntax and operational semantics. This facilitates programming, language implementation and program analysis and transformation.

This combination of language features is not found in other systems programming languages.

The performance of Parlog implementations currently precludes the use of Parlog to program complete operating systems on conventional machines. However, it is possible to point to four areas in which the design and implementation techniques described herein can be applied to advantage.

Programming environments for massively parallel computers. Multiprocessors containing hundreds of processors are already commercially available. Parlog may not be efficient enough to serve as an operating system implementation language for these machines. However, it can be used to implement programming environments that assist application programmers to program them.

Programming environments for computer networks. High-speed local area networks linking together mainframes and workstations already provide many users with access to substantial — and frequently underutilized — computing power. However, the difficulty of distributing applications over such networks means that this resource is rarely exploited. The ease with which Parlog can be executed on loosely coupled machines suggests that it can be used to implement environments that assist programmers in utilizing such resources.

Symbolic computers. Specialized symbolic computers, designed to execute languages such as Prolog, LISP or Parlog at high speeds, require a symbolic systems programming language. For example, McCabe *et al.* [1987] propose an architecture for parallel symbolic computing that links together 64 or more SWIFT symbolic language processors using a high-speed switching network. Parlog is intended to be the systems programming language for this machine.

Process control. Parlog can be used for process control applications in which performance is less critical. Telecommunications researchers, for example, suggest that a Parlog implementation executing at 20K reductions per second is capable of controlling a medium-sized PABX [Robert Virding: personal communication]. This performance can easily be achieved using existing technology. Benefits predicted include increased programmer productivity and ease of maintenance.

8.2 Related Research

Given the current level of interest in logic programming and parallel machines, it is not surprising that considerable research on the use of parallel logic programming languages for systems programming has proceeded concurrently with that reported herein. This work has been referenced in previous chapters where appropriate. This section provides a summary of the most closely related research.

8.2.1 Flat Concurrent Prolog

Flat Concurrent Prolog (FCP) [Mierowsky *et al.*, 1985] is a flat variant of the parallel logic programming language Concurrent Prolog [Shapiro, 1986]. It is the principal language used by a research group at the Wiezmann Institute led by Ehud Shapiro.

Like Parlog, FCP is a development of the Relational Language of Clark and Gregory [1981]. It incorporates guarded clauses, and-parallel evaluation and committed-choice non-determinism. It differs from Parlog in two principal respects. The first is its support for **atomic unification**. FCP permits head unification and guard evaluation to bind process variables. To avoid problems due to incorrect bindings (which may occur if a guard binds variables and subsequently fails), such bindings can only be made accessible to other processes if the clause that generates them is selected to reduce the goal. Head unification and guard evaluation must thus be performed as an atomic operation. Parlog, in contrast, only allows process variables to be bound *after* a clause has been selected to reduce the process (Section 3.2.2). Prior to reduction, clauses are restricted to testing process arguments. Incorrect bindings cannot therefore arise.

The second major difference between FCP and Parlog is its process synchronization mechanism. Unlike Parlog, FCP does not distinguish between input and output arguments. Instead, variables may be annotated as **read-only** occurrences (for example, X?). A process which attempts to bind a read-only occurrence suspends until the variable is bound. For example, consider the FCP query:

```
?- producer(X), consumer(X?).
```

Both producer and consumer may attempt to generate bindings for X. However, as consumer is given a read-only occurrence of X, it suspends until producer instantiates X to a value.

Semantic differences between FCP and Parlog affect both the expressiveness of the languages and their ease of implementation. Atomic unification and the read-only variable permit elegant programming techniques [Shapiro, 1986]. For example, the read-only variable can be used to protect an operating system service against unexpected instantiation of shared variables; this protection must be programmed explicitly in Parlog (Section 4.5). However, these language features complicate language implementation and hence compromise performance. A recent study which benchmarked implementations of Flat Parlog and FCP showed that semantic differences led to significant differences in performance [Foster and Taylor, 1988]. It is hence unclear whether these language features are appropriate in a kernel language.

Research on the use of FCP for systems programming has concentrated on the development of the Logix system [Silverman *et al.*, 1986]. This is a single-user, multitasking programming environment for FCP, written in FCP.

Logix can be regarded as intermediate in scope between the Parlog operating system described in Chapters 4 and 6 and PPS [Foster, 1986], a programming environment that implements the Parlog+ language described in Chapter 7. Like the Parlog operating system, Logix provides task management and basic services such as terminal and disk I/O. However, issues such as exception handling and resource management have not been addressed at the time of writing. Like PPS, it provides the programmer with an augmented user language. The user language supported by Logix is FCP plus modules and a remote procedure call mechanism. This is not as rich as the Parlog+ language supported by PPS, which incorporates metalogical primitives and atomic, serializable transactions.

Logix's treatment of secondary storage is rather different from that of PPS. Secondary storage management is not currently handled by Logix. Instead, users manipulate programs located in Unix files. These may be compiled and loaded as Logix modules. In contrast, PPS's implementation of Parlog+'s persistent state effectively provides its own integrated file system. Programs (modules) can be modified from within PPS using metalogical primitives. An advantage of the Logix approach is that existing Unix tools can be applied to programs. The PPS approach is more sophisticated in that it supports a declarative treatment of program update. It is also arguably a more convincing demonstration of parallel logic languages' ability to handle systems programming tasks.

FCP does not incorporate metacontrol primitives. Instead, as described in Section 5.2.4, task control functions are implemented in Logix using program transformation techniques. The advantages and disadvantages of this approach are discussed in Section 5.7.1. Several levels of control are provided in this way [Hirsch *et al.*, 1987]. Operating system programs are typically executed untransformed and hence uncontrolled. This is the most efficient; however, failure of an untransformed program terminates the entire system. User programs may be transformed so as to provide control functions similar to those provided by the control metacall, on a module-by-module basis. Special mechanisms are required to interface modules with different levels of control [Hirsch, 1987].

Logix's remote procedure call can be used both to execute procedures located in other modules and to make requests to services. It is implemented using a compile-time transformation which translates remote procedure calls into messages on streams. These are routed to the appropriate service or module. Stream caching is used to reduce message routing overheads.

Though currently only executing on uniprocessors, it is planned to port Logix both to multiprocessors and computer networks.

8.2.2 Kernel Language 1

The Japanese Fifth Generation Computer Systems Project [Kawanobe, 1984] aims to develop powerful computer systems based on logic programming languages. The research aims of this project include the development of programming systems for such machines. These programming systems are themselves to be implemented in logic programming languages.

One of the products of the first stage of this project was a sequential Prolog machine (PSI). This featured a quite sophisticated operating system (SIMPOS) [Takagi *et al.*, 1984] implemented in a high-level language, ESP [Chikayama, 1983]. Though it has logic programming components, ESP is essentially an imperative language. Its extensive use of side-effecting primitives means that SIMPOS cannot easily be ported to parallel machines.

More recently, research at ICOT on parallel inference machines [Uchida, 1987] has motivated the definition of the parallel logic programming language KL1 (Kernel Language version 1) [Furukawa *et al.*, 1984]. This language is intended to serve as the principal systems programming language for these machines. Its design was strongly influenced by Parlog and Concurrent Prolog.

KL1 is defined to consist of three components: a user language, KL1-u, implemented on top of a core language, KL1-c, augmented with pragmas for process mapping, process priorities, etc. (KL1-p). At the time of writing, KL1-c is the flat form of the parallel logic language Guarded Horn Clauses [Ueda, 1986] plus a control metacall primitive termed Sho-en. Flat Guarded Horn Clauses is essentially equivalent to Flat Parlog, as Takeuchi and Furukawa [1986] point out, and the Sho-en metapredicate shares many features with the control metacall described herein. KL1-c is thus a very similar language to the kernel language defined in Chapter 5.

KL1 is to be used to implement PIMOS, an operating system for the parallel inference machines currently being developed at ICOT [Sato *et al.*, 1987b]. PIMOS is intended to be a single-user, multi-tasking operating system that will enable users to run applications on these machines. I/O functions are to be provided by a front-end machine such as PSI. PIMOS will no doubt have certain similarities to the Parlog operating system described in Chapters 4 and 6. However, it appears that a rather different approach is to be taken to the implementation of lower-level functions such as distributed metacontrol resource management. These will be implemented in the language kernel rather than in KL1. The advantages and disadvantages of this approach are discussed in Section 5.7.2.

8.3 Future Research

There is considerable scope for further research in a number of areas related to the use of Parlog as a systems programming language.

Parlog implementation. Perhaps the most pressing area for future research is efficient implementation of the kernel language. The implementation techniques described in Chapter 5 require further development. Experimental studies are required to quantify the costs of the various distributed unification algorithms proposed. Investigation of improved memory management techniques is particularly important.

Support for application development. The research reported herein has shown how fundamental problems in operating system and programming environment design can be treated in Parlog. Future research can now focus on developing programming environments and tools that aid the programmer to develop applications and exploit parallelism. This research can influence operating system and language design.

Formal semantics. A formal semantics for Parlog is required if Parlog system programs are to be verified with respect to global properties such as termination and freedom from deadlock.

Robust systems. This study has assumed failsafe components and reliable communications. Further research must consider the implications of unreliable components and communications for language and system design. Parlog's control metacall and exception mechanism can be used to localize and report errors. Further language support may be required if Parlog is to be used to program robust distributed systems.

Programming in the large. If Parlog is to be used to program large systems, support is required for programming in the large. It must be possible to define *modules* [Parnas, 1972] which communicate with other modules (and with the external world) in certain well-defined ways. As noted in Section 4.5, Parlog programs are to an extent naturally modular: there are no global structures and processes can only communicate through shared variables. However, as variables are themselves global entities, the propagation of variables can result in complex, unexpected interactions between processes. The validation techniques presented in Section 4.5 can be used to avoid these problems by restricting the sharing of logical variables between modules and copying variable bindings across module interfaces.

Support for programming in the large in Parlog can perhaps most easily be achieved by providing (a) a configuration language, used to specify modules and interfaces, and

(b) a preprocessor which enforces interface specifications by transforming programs so as to apply techniques described in Section 4.5.

250

Appendix I.

The Control Metacall — A Specification.

This appendix presents an executable specification for the three-argument control metacall described in Section 3.4, extended with the exception status message described in Section 4.2.2. Recall that a call to the control metacall has the general form: call(G,S,C) and initiates execution of G using an implicit program. Execution of G can then be controlled by instantiating the control variable C to the constant stop to abort execution or to a list with head suspend or continue to suspend or resume execution. The status variable S is unified with the constants succeeded, failed or stopped, to report successful termination, failure or abortion respectively. It is also unified with a list with head suspend or continue to report that processing of a suspend or continue command has completed. Finally, as described in Section 4.2.2, the status message exception(T,G,C) reports that goal G cannot be solved, for a reason indicated by the term T. C is a metavariable that can be instantiated to provide a new goal to replace G in the computation.

The specification takes the form of an enhanced metainterpreter for the flat subset of Parlog (Section 5.2.1). A metainterpreter is an interpreter for a language, written in the language that it interprets. An enhanced metainterpreter incorporates additional code that (in this case) monitors and controls execution of a query that is being interpreted.

A system primitive (or user-defined procedure) clause(G,B) is assumed in this specification. This defines the program with respect to which interpreted queries are evaluated. clause(G,B) has the logical reading: a goal G can be reduced to body B. A call clause(G,B) evaluates the guards of clauses in the procedure defining G and returns the body B of a clause that can be used to reduce G. It binds B to raise_exception(T,G) if G cannot be reduced. If G is defined, but reduction fails, T = failure. T takes other values if reduction fails for other reasons (for example, because G invokes an undefined relation, or is an illegal call to a primitive).

Two simple examples are first presented to introduce Parlog metainterpreters. The first, Program I.1, is a trivial metainterpreter that interprets Flat Parlog programs but does not provide monitoring or control functions. This is a Flat Parlog version of a FCP metainterpreter presented by Safra and Shapiro [1986]. A call reduce(G) has the same logical reading as a call to its argument. The first clause of reduce reads: the goal true is true. The second reads: a conjunction of two goals A and B is true if A and B

are true. The third reads: a goal G is true if it can be reduced to a body B and B is true.

```
mode reduce(Goal?).

reduce(true).
reduce((A,B)) ← reduce(A), reduce(B).
reduce(G) ←
        G =/= true, G =/= (_,_), G =/= raise_exception(_,_) · clause(G,B), reduce(B).
```

Program L1 Simple Parlog metainterpreter.

Such a metainterpreter can be enhanced to report termination, permit control and so forth. This is achieved by:

- Providing shared variables to link the reduce processes representing the processes in a task, so that a central monitor can communicate with all such processes.
- Extending the procedure defining reduce so that it reports changes in the state of a process that it is reducing to the central monitor.
- Extending the procedure defining reduce so that it responds to control messages received from the central monitor (to suspend, resume, abort its process).

```
mode call(Goal?, Status↑), s_reduce(Goal?, Left?, Right↑), monitor(Left↑, Right?, Status↑).

call(G, S) ← s_reduce(G, L, R), monitor(L, R, S).                          (C1)

monitor(L, R, succeeded) ← L == R : true.                                  (C2)

s_reduce(true, L, L).                                                      (C3)
s_reduce((A,B), L, R) ← s_reduce(A, L, M), s_reduce(B, M, R).              (C4)
s_reduce(G, L, R) ←                                                        (C5)
        G =/= true, G =/= (_,_), G =/= raise_exception(_,_) :
        clause(G,B), s_reduce(B, L, R).
```

Program L1 Termination detecting Parlog metainterpreter.

The simplest way of linking processes is in a *circuit*. Each reduce process possesses an input and an output stream, accepts control messages on its input stream and generates status messages on its output stream. The streams are used to link all reduce processes in a ring. Each process forwards messages from its input to its output. Messages hence flow around the circuit, controlling processes and signalling changes in process status.

Program I.2 illustrates the use of a circuit and illustrates how it can be used to detect termination of an interpreted goal.

call(G,S) has the logical reading: G is true and S=succeeded. A call call(G,S) evaluates G and, if and when evaluation of G terminates, unifies S with the constant succeeded. Processes created during evaluation of G are linked in a circuit using their second and third arguments (C4). Termination detection is achieved using a programming technique due to Takeuchi [1983] called the **short circuit**. (For another example of the application of this technique, see Section 6.1). Each process unifies its second and third arguments — and hence closes its part of the circuit — upon successful termination (C3). The monitor process has references to the two ends of the circuit. These will be identical variables precisely when all s_reduce processes — and hence the goal these processes are interpreting — have terminated. monitor can then bind S to succeeded (C2).

Program I.3 presents the full specification of the Parlog control metacall. Points to note include:

- A monitor (mon) is associated with a task. This coordinates its monitoring and control. An initial call to Program I.3, call(G,S,C), creates a monitor process and an initial reduce process to interpret the goal G. reduce processes are linked in a circuit, as in Program I.2. The circuit is used for control and to report exceptions as well as for termination detection.
- The specification reports *process failure*. The reduce procedure is made failsafe by adding a clause that handles exceptions (C5). Exceptions are forwarded on the circuit. Other reduce processes forward exceptions (C7). mon thus receives exception messages on the right-hand side of its circuit. It reports exceptions denoting process failure as failure, and aborts the task (C15); other exceptions are output on the task's status stream (C16).
- The control metacall's *exception message* (Section 4.2.2) is implemented by forwarding an exception message that includes a continuation variable C on the

circuit and reducing to a call to this variable (C5). This call suspends until C is bound.

The specification deals with *control messages*. An extra clause for reduce (C7) handles control messages received on the circuit. reduce calls control, which processes the message and then forwards it. A stop message causes control to terminate (C9); suspend causes it to wait for another control message (C10); continue causes it to call reduce and hence resume execution (C11). exception messages are forwarded on the circuit (C12).

The central monitor (mon) deals with control messages received on a task's control stream by generating a control message on the left-hand side of the circuit (C17,18). Only when this message is received on the right-hand side of the circuit, indicating that all processes have been controlled, does mon echo the corresponding status message (stopped : C14; suspend, continue : C16).

mon detects changes in the status of the task it is monitoring and signals these on the task's status stream (C13-16). A closed circuit indicates termination (C13). mon's handling of failure and exceptions has been noted.

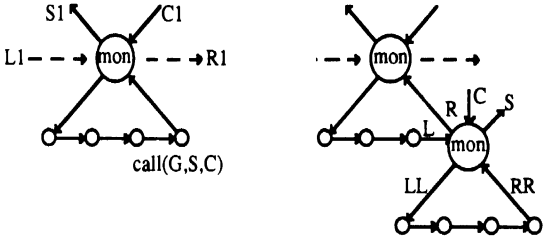
The specification handles nested metacalls. An extra clause for reduce handles calls to call/3 (C6). A call reduce(call(G,S,C),L,R) reduces to a recursive call to reduce, with new status and control variables, and a new monitor, which monitors and controls the new task. The tree structure that arises when metacalls are nested is thus implemented using a flat pool of monitor processes connected using shared variables (see Figure I.1).

The circuit linking the processes representing a task may thus link both reduce and mon processes, representing processes and tasks respectively. As has been noted, a monitor handles control messages directed to its task (C17,18) and changes in its task's status (C13-16). It also handles control messages directed to the task of which *it* is a part. Control messages received on its parent's circuit cause it to abort, suspend or resume its task (C19-24).

Control of nested tasks is implemented correctly. Controlling a parent controls any offspring tasks (that is, any tasks in its circuit). But suspending and then resuming a parent with suspended offspring leaves the offspring suspended. This is achieved by associating a status with each monitor. This takes values suspend and continue (it is initially continue) and is set whenever a task is controlled (C18). A control message only affects a subtask if its status is continue (C20,23). Otherwise the message is forwarded immediately (C21,22).

- The specification uses a metalogical var test in one clause (C6). This can be ignored in the logical reading of the metainterpreter. It is not necessary for the metainterpreter's correct execution, but ensures that clause C8 is used to reduce a process in preference to clause C4 if a control message is pending on the task's circuit. It hence ensures that a task is controlled as soon as possible after a control message is placed on the left-hand side of its circuit.

Figure I.1 illustrates the process network before and after a new task is created by a nested call to the control metacall.



(a) Before metacall. (b) After metacall.

Figure I.1 Nested metacalls.

In Figure I.1 (a), the reduce processes representing a task are linked together in a circuit. A monitor mon supervises the execution of these processes. mon takes as arguments left and right streams (L1, R1) to the circuit of the task of which it is a member (the initial task is not a member of any task); left and right streams to the circuit of the task it is monitoring; this task's status and control variables (S1, C1) and a state, not shown.

If a process in the task reduces to a call to the control metacall, call(G,S,C), then a new monitor and circuit are created, as shown in Figure I.1 (b). The new monitor takes as arguments left and right streams to its parent's circuit (L, R); left and right streams to the new task's circuit (LL, RR); the new task's status and control variables (S, C) and a state, initially continue.

```
mode call(Goal?, Status↑,Control?).

call(G,S,C) ← reduce(G,L,R), mon(_,_,L,R,S,C,continue).                    (C1)

mode reduce(Goal?, Left?, Right↑).

reduce(true, L, L).                                  % true: succeeds.      (C3)
reduce((A,B), L, R) ← reduce(A, L, M), reduce(B, M, R).  % Reduce conjuncts. (C4)
reduce(call(G,S,C), L, R) ←                          % Nested metacalls:    (C6)
      reduce(G, LL, RR), mon(L, R, LL, RR, S, C, continue).% ... create subtask.
reduce(G, L, R) ← var(L), G =/= (_,_), G =/= true,   % Other goals:        (C6)
      G =/= raise_exception(_,_), G =/= call(_,_,_) :  %   find &
      clause(G, B), reduce(B, L, R).                   %   reduce body.
reduce(raise_exception(T,G), L, [exception(T,G,C) |R]) ←  % Signal exception, (C5)
      reduce(C, L, R).                               % ... and continue.
reduce(G, [exception(T,G,C) |L], [exception(T,G,C) |R]) ←  % Exception message: (C7)
      control(G, L, R).                              %   forward in circuit.
reduce(G, [M |L], R) ←                               % Control message:     (C8)
      M =/= exception(_,_,_) : control(G, [M |L], R).  %   control process.

mode control(Goal?, Left?, Right↑ ).

control(G, [stop |L], [stop |L]);                    % Terminate process.   (C9)
control(G, [suspend | L], [suspend | R]) ← control(G, L, R)  % Suspend process. (C10)
control(G, [continue | L], [continue | R]) ← reduce(G, L, R).  % Resume process. (C11)
control(G, [M | L], [M | R]) ←                       % Forward exception.  (C12)
      M == exception(_,_,_) : reduce(G, L, R)

mode mon(L?, R↑, LL↑, RR?, S↑, C?, State?).

mon(L, L, LL, RR, succeeded, C, St) ← LL == RR : true.  % Subtask terminates. (C13)
mon(L, L, LL, [stop |RR], stopped, C, St).           % Subtask aborted.    (C14)
mon(L, L, [stop |LL], [exception(failure,_,_) |RR], failed, C, St). % Subtask failed. (C15)
mon(L, R, LL, [M |RR], [M |S], C, St) ←              % Subtask status      (C16)
      M =/= stop, M =/= exception(failure,_,_) :     %   message:
      mon(L, R, LL, RR, S, C, St).                   %   output on status.
mon(L, R, [stop |LL], RR, S, stop, St) ←             % Subtask control:    (C17)
      mon(L, R, LL, RR, S, C, St).                   %   stop subtask.
mon(L, R, [M |LL], RR, S, [M |C], St) ←              % Subtask control:    (C18)
      mon(L, R, LL, RR, S, C, M).                    %   suspend/continue.
mon([stop |L], R, [stop |LL], RR, S, C, St) ←        % Parent task control: (C19)
      terminate (L, R, LL, RR).                      %        stop.
mon([suspend |L], R, [suspend |LL], RR, S, C, continue) ←  % Parent task control: (C20)
      echo(suspend, L, R, LL, RR, S, C, continue).   %   suspend subtask.
mon([suspend |L], [suspend |R], LL, RR, S, C, suspend) ←  % Parent task control: (C21)
```

```
        mon(L, R, LL, RR, S, C, suspend).                    % already suspended.
mon([continue |L], [continue |R], LL, RR, S, C, suspend) ← % Parent task control: (C22)
        mon(L, R, LL, RR, S, C, suspend).                    % already suspended.
mon([continue |L], R, [continue |LL], RR, S, C, continue) ← % Parent task control: (C23)
        echo(continue, L, R, LL, RR, S, C, continue).        % resume subtask.
mon([M |L], [M |R], LL, RR, S, C, St) ←                      % Other parent status: (C24)
        mon(L, R, LL, RR, S, C, St).                         % forward on circuit.

mode terminate(L?, R↑, LL↑, RR?).

terminate (L, [stop |L], LL, [stop |RR]).  % Forward stop when subtask terminates. (C25)
terminate (L, R, LL, [M |RR]) ← terminate (L, R, LL, RR).  % Ignore other status. (C26)

mode echo(M?, L?, R↑, LL↑, RR?, S↑, C?, State?).

echo(M, L, [M |R], LL, [M |RR], S, C, St) ← mon(L, R, LL, RR, S, C, St).          (C27)
echo(M, L, R, LL, [M1 |RR], [M1 |S], C, St) ←                                    (C28)
        M1 =/= M, M1 =/= stop, M1 =/= exception(failure,_,_) :
        echo(M, L, R, LL, RR, S, C, St).
echo(M, L, [M |L], LL, [stop |RR], stopped, C, St).                              (C29)
echo(M, L, [M |L], [stop |LL], [exception(failure,_,_) |RR], failed, C, St).     (C30)
```

Program I.3 Executable specification for control metacall.

258

Appendix II.

A Comparison of Two Approaches to Metacontrol

Two approaches to the implementation of metacontrol mechanisms in parallel logic languages have been proposed. One is based on kernel support for metacontrol functions (Section 5.2.2). The other uses *transformation* to generate programs that can be controlled (Section 5.2.4). This appendix describes an experimental study that compares the efficiency of these two approaches.

II.1 Method

Any metacontrol mechanism necessarily introduces some run-time overhead. This may take the form of increased code size, CPU time requirements and run-time memory requirements. It may affect two classes of program:

- programs that do *not* form part of a controlled task (*uncontrolled* programs).
- programs that *do* form part of a controlled task (*controlled* programs).

For example, an operating system may consist of an uncontrolled system program that initiates, monitors and controls application programs. If control is provided by kernel mechanisms, both the operating system and application programs incur the same overhead, as both are executed using the same kernel mehanisms. If control is provided by transformation, only transformed application programs incur overhead. Table II.1 illustrates this.

	$Overhead_{uncontrolled}$	$Overhead_{controlled}$
Kernel Support	C_1	C_1
Transformation	0	C_2

Table II.1 Implementation costs of metacontrol functions.

The values C_1 and C_2 can be most easily determined by experimental studies. Some figures for C_2 are given by Hirsch *et al.* [1986]; these are repeated below. It is more useful however to obtain figures for C_1 and C_2 using the same implementation, to minimize discrepancies due to language and implementation differences.

II.1.1 Levels of Control

For the purposes of evaluation, it is useful to isolate the various control functions described by the metacontrol primitives presented in Section 5.2.2. This provides a number of control levels, each generally requiring mechanisms provided by lower levels for its implementation. The levels are named *0* to *3* and are characterized in Table II.2.

i	*Description*
0	No controls.
1	Failure and exceptions reported in a task: S = [exception(_,_,_) |_]
2	Task may be suspended, resumed and aborted: SUSPEND(TR), ..., STOP(TR).
3	Successful task termination is reported: S = succeeded.

Table II.2 Levels of control.

Level 0 provides no additional functionality: it implements the process pool model of computation of the basic language (Section 3.2.2). *Level 1* introduces the sub-pool or **task**. Every process is part of a task and process failure in a task is reported. This provides support for the status message exception(_,_,_). *Level 2* in addition permits a task to be suspended, resumed and aborted; it supports the SUSPEND, CONTINUE and STOP primitives. *Level 3* provides for reporting of successful termination of a task: the status message succeeded. Level 3 is effectively the level of control provided by the TASK primitive (Section 5.2.2).

II.1.2 Implementations

Gregory *et al.* [1988] describe a parallel logic language implementation that provides kernel support for metacontrol functions. This is the *Sequential Parlog Machine* (SPM). The SPM implements the And-Or tree model of Parlog execution [Gregory, 1987], which represents a Parlog computation as a tree of processes. A task is merely a new type of node in this tree: existing mechanisms for signalling termination and aborting subtrees require little modification to implement metacontrol functions. It is thus not surprising that metacontrol overheads appear to be insignificant in the SPM. Studies show however that the SPM's And-Or tree model is more expensive to implement than Flat Parlog's process pool model [Foster and Taylor, 1988]. Some of this cost is due to mechanisms which can be used to implement metacontrol functions. The SPM cannot therefore be used for comparative studies of metacontrol mechanisms.

The Flat Parlog machine described in Section 5.3 provides a suitable basis for comparative studies of metacontrol mechanisms. This abstract machine is based on the process pool model of computation and hence provides no inherent support for task control. An implementation of this machine (an emulator, written in the C programming language) has been extensively instrumented and has been used previously for quantitative comparisons [Foster and Taylor, 1988]; its behaviour and structure are well understood. Finally, this implementation executes Flat Parlog particularly efficiently.

The Flat Parlog machine, henceforth referred to as FP_0, was progressively extended to implement the control levels *1* to *3* described above, yielding new machines FP_1, FP_2 and FP_3. This permitted precise measurement of the costs incurred when implementing each control level. The extensions to FP_0 required to implement machines FP_1, FP_2 and FP_3 are described in detail in [Foster, 1987b]. The extensions required to implement FP_3 are essentially those described in Section 5.4. However, to permit comparison with transformations defined by Hirsch *et al.*, FP_3 does not support the Parlog task scheduler described in Section 5.4. Instead, a simple round-robin task scheduling algorithm is implemented in the machine itself.

To permit benchmarking of the transformation approach, three transformations described by Hirsch *et al.* [1986] were implemented. These transformations implement control levels 1, 2 and 3 and are termed here T_1, T_2 and T_3.

II.1.3 Benchmark Programs

Four benchmark programs were selected. To facilitate comparison with the results reported by Hirsch *et al.*, similar programs were used. *Reverse* performs naive $O(n^2)$ reverse of a list of 32 elements, 32 times. *Primes* generates all primes less than 1000 using the parallel sieve of Eratosthenes. *QSort* applies the quicksort sorting algorithm to a 1000 element list. These programs are given at the end of this appendix. The last benchmark program, *Compiler*, is the program that translates parsed Flat Parlog programs into sequences of Flat Parlog machine instructions. This program is run with the Flat Parlog assembler as data.

II.1.4 Summary

Four levels of control (0 to 3), four machines (FP_0-FP_3) and three transformations (T_1-T_3) were defined. Four benchmark programs were selected. Experiments were then performed to determine the overheads incurred when the benchmark programs

were executed on each of the machines FP_0, ..., FP_3 and, when transformed using transformations T_1, T_2 and T_3, on FP_0. CPU time, code size and run-time memory requirements were measured.

II.2 Previous Results

Hirsch *et al.* [1986] present benchmark figures indicating the space and CPU time overheads associated with transformations T_1, T_2 and T_3. These figures are summarized in Table II.3. The benchmark programs *Reverse* and *Primes* are as described above. For the *Compiler* benchmark, Hirsch *et al.* use an FCP code generator compiling the primes program. This program is comparable to the Flat Parlog compiler used for the benchmarks reported here.

	Performance: T_0 / T_i				Code Size: T_0 / T_i			
	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3
Reverse	1.00	1.10	1.42	1.65	1.00	1.57	2.47	3.18
Primes	1.00	1.07	1.26	1.37	1.00	1.47	2.23	2.87
Compiler	1.00	1.03	1.06	1.16	1.00	1.41	2.08	2.42

Table II.3 Transformation overheads, as reported by Hirsch *et al.*

II.3 New Results

Each benchmark program was run on each of the FP_i and the speed, in reductions per second (RPS), determined. These results are presented in Table II.4. All performance figures are for the mean of 100 runs, excluding garbage collection, on a SUN-3/75 workstation.

	FP_0	FP_1	FP_2	FP_3
Reverse	5991	5768	5924	5975
Primes	2579	2516	2543	2560
QSort	2584	2529	2541	2556
Compiler	2114	2114	2070	2075

Table II.4 Kernel support performance (RPS, FP_0—FP_3).

The transformations T_1, T_2 and T_3 were applied to each of the benchmark programs and the transformed programs were run on FP_0. Performance figures are presented in Table II.5. This table also gives the code size of the transformed

programs.

	Reductions per second				Code size (bytes)			
	T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3
Reverse	5991	5793	5107	4745	844	1252	1892	1992
Primes	2579	2494	2208	2158	644	1228	1928	2028
QSort	2584	2385	2216	2140	1880	2504	3396	3708
Compiler	2114	2035	1965	1835	27536	32364	50108	54464

Table II.5 Transformation performance and code size (FP_0).

Additional run-time memory requirements due to both kernel support and transformation were also measured, but were not found to be a significant source of overhead.

II.4 Discussion

Table II.6 summarizes the benchmark results, showing performance degradation associated with kernel support and transformation for control levels 1 to 3, and the increase in code size associated with transformations T_1, T_2 and T_3.

	Perf: Kernel.			**Perf: Trans.**			**Code: Trans.**		
	FP_1	FP_2	FP_3	T_1	T_2	T_3	T_1	T_2	T_3
Reverse	1.04	1.01	1.00	1.03	1.17	1.26	1.48	2.24	2.36
Primes	1.02	1.01	1.00	1.03	1.16	1.19	1.90	2.99	3.15
QSort	1.02	1.02	1.01	1.13	1.17	1.21	1.33	1.81	1.97
Compiler	1.00	1.02	1.02	1.04	1.08	1.15	1.18	1.81	1.98

Table II.6 Comparative performance and code size.

These results indicate that kernel support only introduces small overheads on uniprocessors. In the most complex benchmark, *Compiler*, the performance difference between FP_3 and FP_0 is 2%. Code size is of course the same. Study of the modifications to the Flat Parlog abstract machine required to support kernel mechanisms (described in Section 5.4) indicates why kernel support is so inexpensive. Overheads are only incurred at certain points in the computation process, such as process creation, process termination and process switching. Overheads are generally small, consisting of simple operations on registers. Most significantly, the notion of **current task** is introduced. Tasks are reduced in turn using (for these benchmarks) a round-robin strategy. The contents of a current task's task record is buffered in global

registers. This ensures that overheads associated with task selection are only incurred in the rare event of a switch to another task. (For the benchmarks, this was performed once every 500 reductions; in a Parlog machine, this could be triggered by a timer interrupt). Otherwise, reduction proceeds almost as efficiently as in a machine that does not support tasks.

It will be observed that performance overheads do not increase uniformly across the FP_i with increasing i. For example, *Primes* suffers a 2% performance degradation on FP_1 but virtually none on FP_3. This is due to reasons unconnected with the functionality of the FP_i. Because performance differences are so small, machine-dependent factors related to the swapping behaviour of the different implementations become significant. These factors hinder precise comparisons of the various levels of kernel support. They do not however invalidate the conclusion that kernel support only incurs low overhead.

The benchmark results for transformation generally indicate lower overheads than those reported by Hirsch *et al.* This is made clear in Table II.7, which presents both sets of figures for performance and code size. These lower overheads may be due to differences in both compilation techniques and in the language implementations used to perform the benchmarks. The Flat Parlog machine (FP_0) used to benchmark the transformations provides a particularly efficient implementation of test primitives var and data introduced by transformation. Nevertheless, the results reported here still indicate that transformation leads to significant time and space overheads on the implementations studied.

	Reductions per second (T_0 / T_i)						Code Size (T_0 / T_i)					
	Hirsch *et al.*			Foster			Hirsch *et al.*			Foster		
	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
Reverse	1.10	1.42	1.65	1.03	1.17	1.26	1.57	2.47	3.18	1.48	2.24	2.36
Primes	1.07	1.26	1.37	1.03	1.16	1.19	1.47	2.23	2.87	1.90	2.99	3.15
Compiler	1.03	1.06	1.16	1.04	1.08	1.15	1.41	2.08	2.42	1.18	1.81	1.98

Table II.7 Transformation overheads reported by Hirsch *et al.* and Foster.

In summary, referring to the most substantial benchmark, *Compiler*, and the most comprehensive level of control, Level 3:

- the CPU cost of kernel support is 2 %: this cost is incurred whether or not the program is to be controlled

- the CPU cost of transformation is 15 %: this cost is not incurred if the program is not to be controlled
- kernel support does not effect code size
- transformation increases code size 98 %
- run-time memory requirements are not a significant source of overhead

A potential disadvantage of kernel support is that its overheads are incurred by all programs, whether controlled or not. The overheads associated with kernel support are however so low that this is not considered important.

The higher overheads associated with transformation can be reduced using better compilation techniques. Kernel support overheads represent at the very least an upper bound on the improvements that can be expected, as any optimization introduced by sophisticated compilation techniques can always be incorporated in a kernel mechanism.

II.5 The Benchmark Programs

The benchmark programs *Reverse*, *QSort* and *Primes* follow.

Reverse

```
mode reverse(Xs?, Ys↑), append(Xs?, Ys?, Zs↑)

reverse([X |Xs], Ys) ← reverse(Ys, Zs), append(Zs, [X], Ys).
reverse([ ], [ ])

append([X |Xs], Ys, [X |Zs]) ← append(Xs, Ys, Zs).
append([ ], Ys, Ys).
```

Primes

```
mode primes(Ps↑, Limit?), integers(Ns↑, Limit?), integers(N?, Ns↑, Limit?), sift(Ns?, Ps↑),
        filter(Num?, M?, Ms?,N↑).

primes(Ps, Limit) ← integers(Ns, Limit), sift(Ns, Ps).

integers(Ns, Limit) ← integers(2, Ns, Limit).

integers(N, [N |Ns], Limit) ← N < Limit, N1 is N + 1 : integers(N1, Ns, Limit).
integers(N, [ ], Limit) ← N >= Limit : true.

sift([N |Ns], [N |Ps]) ← filter(N, N, Ns, Ns1), sift(Ns1, Ps).
sift([ ], [ ]).

filter(Num, M, [M |Ms], Ns) ← filter(Num, M, Ms, Ns).
filter(Num, N, [M |Ms], Ns) ← M > N, N1 is N + Num · filter(Num, N1, [M |Ms], Ns).
filter(Num, N, [M |Ms], [M |Ns]) ← N < M : filter(Num, N, Ms, Ns).
filter(_, _, [ ], [ ]).
```

Quicksort

```
mode qsort(Xs?, Ys↑), qsort(Xs?, L↑, R?), part(X?, Ys?, Us↑, Vs↑).

qsort(Xs, Ys) ← qsort(Xs, Ys, [ ]).

qsort([X | Xs], L, R) ← part(X, Xs, Us, Vs), qsort(Us, L, [X |M]), qsort(Vs, M, R).
qsort([ ], L, L).

part(X, [Y |Ys], [Y |Us], Vs) ← X > Y : part(X, Ys, Us, Vs).
part(X, [Y |Ys], Us, [Y |Vs]) ← X <= Y : part(X, Ys, Us, Vs).
part(_, [ ], [ ], [ ]).
```

References

Abramsky, S. 1982. A simple proof theory for non-determinate recursive programs. Research report, Queen Mary College, Computer Systems Laboratory.

Agha G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass.

Andrews, G.R. and Schneider, F.B. 1983. Concepts and notations for concurrent programming. In *Computing Surveys*, 15(1), 3-43.

Aoyagi, T., Fujita, M. and Moto-oka, T. 1985. The temporal logic programming language Tokio. In *Proc. 4th Logic Programming Conf.* (Tokyo), Springer-Verlag LNCS-221, 128-137.

Arvind and Brock, J.D. 1982. Streams and managers. In *Operating Systems Engineering*, Springer-Verlag LNCS-143, 452-465.

Ashcroft, E.A. and Wadge, W.W. 1976. LUCID - a formal system for writing and proving programs. In *SIAM J. Computing*, 5(3).

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R. 1983. An approach to persistent programming. In *Computer Journal*, 26(4), 360-365.

Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of processes. In *CACM* 21(8), 613-641.

Badal, D.Z. 1979. Correctness of concurrency control and implications in distributed databases. In *Proc. COMPSAC '79 Conf.*, Chicago.

Bernstein, P.A., Goodman, N. and Hadzilacos, V. 1983. Recovery algorithms for database systems. In *Information Processing 83; Proc. IFIP 9th World Computer Congress*, Amsterdam: North-Holland, 799-807.

Bowen, K.A., and Kowalski, R.A. 1982. Amalgamating language and metalanguage in logic programming. In *Logic Programming*, Academic Press, pp.153-172.

Bowen, K.A. 1985. Meta-level programming and knowledge representation. In *New Generation Computing*, 3(4), 359-383.

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. In *IEEE Transactions on Software Engineering* SE-1(2), 199-207.

Brinch Hansen, P. 1976. The SOLO operating system. In *Softw. P & E*, 6, 141-149.

Brinch Hansen, P. 1987. Joyce - a programming language for distributed systems. In *Softw. P & E*, 17(1), 29-50.

Broy, M. 1983. Applicative real-time programming. In *Information Processing 83; Proc. IFIP 9th World Computer Congress*, Amsterdam: North-Holland, 259-264.

Burstall, R.M. and Darlington, J. 1977. A transformation system for developing recursive programs. In *JACM*, 24(1), 44-67.

Burstall, R.M., MacQueen, D.B. and Sannella, D.T. 1980. HOPE: an experimental applicative language. Internal Report CSR 62-80, Dept of Comp. Sci., University of Edinburgh.

Chikayama, T. 1983. ESP — Extended Self-contained Prolog — as a preliminary kernel language of fifth generation computers. In *New Generation Computing*, 1(1), 11-24.

Church, A. 1941. The calculi of lambda conversion. In *Annals of Mathematics Studies*, 6. Princeton University Press, Princeton, NJ.

Clark, K.L. 1978. Negation as failure. In Logic and Databases, New York: Plenum Press, 293-322.

Clark, K.L. and Gregory, S. 1981. A relational language for parallel programming. In *Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architectures*, 171-178.

Clark, K.L. and Gregory, S. 1984. Notes on systems programming in PARLOG. In *Proc. Intl Conf. on 5th Generation Computer Systems*, Amsterdam: North-Holland, 299-306.

Clark, K.L. and Gregory, S. 1986. PARLOG: parallel programming in logic. In *ACM Trans. Program. Lang. Syst.*, 8 (1), 1-49.

Clark, K.L. and McCabe, F.G. 1979. The control facilities of IC-Prolog. In *Expert Systems in the Micro-electronic Age*, Edinburgh University Press.

Conery, J.S. and Kibler, D.F. 1981. Parallel interpretation of logic programs. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, New York: ACM, 163-170.

Conway, M.E. 1963. A multiprocessor system design. In *Proc. AFIPS Fall Jt Computer Conf.*, Spartan Books, Baltimore, Maryland, 139-146.

Crammond, J.A. 1985. A comparative study of unification algorithms for Or-parallel execution of logic programs. In *IEEE Trans. on Computers*, C-34(10), 911-917.

Crammond, J.A. 1988. *Parallel Implementation of Committed-choice Languages*, PhD thesis, Herriot-Watt University, Edinburgh.

Darlington, J., Field, A.J. and Pull, H. 1985. The unification of functional and logic languages. In *Logic Programming: Relations, Functions and Equations*, Prentice-Hall, Englewood Cliffs, NJ.

Darlington, J. and While, L. 1986. The imposition of temporal constraints on the execution of term-rewriting systems. Unpublished report, Dept of Computing, Imperial College, London.

Darlington, J. and Reeve, M. 1981. ALICE, a multiprocessor reduction machine for the parallel evaluation of applicative languages. Research report, Dept of Computing, Imperial College, London.

DeGroot, D. 1984. Restricted And-parallelism. In *Proc. Intl Conf. on 5th Generation Computer Systems*, Amsterdam: North-Holland, 471-478.

Dennis, J.B. and van Horn, E.C. 1966. Programming semantics for multiprogramming computations. In *CACM*, 9(3), 143-155.

Dijkstra, E.W. 1968a. The structure of the 'THE'-multiprogramming system. In *CACM*, 11(5), 341-346.

Dijkstra, E.W. 1968b. Cooperating sequential processes. In *Programming Languages*, New York: Academic Press.

Dijkstra, E.W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. In *CACM*, 18(8), 453-457.

Dijkstra, E.W., Feijen, W.H.J. and van Gasteren, A.J.M. 1983. Derivation of a termination detection algorithm for distributed computations. In *Information Processing Letters*, 16(5), North-Holland, 217-219.

DoD, 1982. Reference manual for the Ada programming language. ANSI/MIL-STD-1815A, United States Department of Defence.

van Emden, M.H. and de Lucena Filho, G.J. 1982. Predicate logic as a language for parallel programming. In *Logic Programming*, Academic Press.

Foster, I.T. 1986. Parlog Programming System: user guide and reference manual. Research report, Dept of Computing, Imperial College, London.

Foster, I.T. 1987a. Logic operating systems: design issues. In *Logic Programming: Proc. 4th Intl Conf.*, MIT Press, 910-926.

Foster, I.T. 1987b. Efficient metacontrol in parallel logic languages. Research report, Department of Computing, Imperial College, London.

Foster, I.T. 1988. An experiment in the use of Parlog for real-time process control. Research report, Department of Computing, Imperial College, London. In preparation.

Foster, I.T., Gregory, S., Ringwood, G. A., and Satoh, K. 1986. A sequential implementation of PARLOG. In *Proc. of the 3rd Intl. Logic Programming Conf.*, LNCS-225, New York: Springer-Verlag, 149-156.

Foster, I.T. and Kusalik, A.J. 1986. A logical treatment of secondary storage. In *Proc. IEEE Symp. on Logic Programming*, Salt Lake City, 58-69.

Foster, I.T. and Taylor, S. 1988. Flat Parlog: a basis for comparison. In *International Journal of Parallel Processing*, 16(2), in press.

Friedman, D.P. and Wise, D.S. 1976. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, Edinburgh University Press, 257-284.

Friedman, D.P. and Wise, D.S. 1979. An approach to fair applicative multiprogramming. In *Semantics of Concurrent Computations*, Springer-Verlag.

Friedman, D.P. and Wise, D.S. 1980. An indeterminate constructor for applicative programming. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, Springer-Verlag.

Furukawa, K., Kunifuji, S., Takeuchi, A. and Ueda, K. 1984. The conceptual specification of the Kernel Language version 1. Technical report TR-054, ICOT, Tokyo.

Glasgow, J.I. and McCewan, G.H. 1987. The development and proof of a formal specification for a multilevel secure system. In *ACM Trans. Comp. Syst.* 5(2), 151-184.

Goguen, J.A. and Meseguer, J. 1984. Equality, types, modules and generics for logic programming. In *Proc. 2nd Intl Logic Programming Conf.*, 115-125.

Gregory, S. 1987. *Parallel Logic Programming in PARLOG*. Reading, Mass.: Addison-Wesley.

Gregory, S., Foster, I.T., Burt, A.D. and Ringwood, G.A. 1986. An abstract machine for the implementation of PARLOG on uniprocessors. In *New Generation Computing*, in press.

Halstead, R.H. and Loaiza, J.R. 1985. Exception handling in Multilisp. In *Proc. 1985 Intl Conf. on Parallel Processing*, St Charles, Ill., 822-830.

Harrison, D. 1987. RUTH: a functional language for real-time programming. In *Proc. PARLE Conf.*, Springer -Verlag LNCS-258, 297-314.

Henderson, P. 1980. *Functional Programming: Application and Implementation*, London: Prentice-Hall.

Henderson, P. 1982. Purely functional operating systems. In *Functional Programming and its Applications*, Cambridge University Press.

Henderson, P. and Morris, J.H. 1976. A lazy evaluator. In *Proc. 3rd ACM Symp. on Principles of Programming Language*, Atlanta, Georgia.

Hermenigildo, M.V. 1986. An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel. TR-86-20, Dept of Computer Sciences, University of Texas at Austin.

Hewitt, C.E. 1977. Viewing control structures as patterns of passing messages. In *J. of Artificial Intelligence*, 8(3), 323-364.

Hirsch, M. 1987. *The Logix System*. MSc thesis, Weizmann Institute, Rehovot.

Hirsch, M., Silverman, W., and Shapiro, E. 1986. Layers of protection and control in the Logix system. Technical report CS-20, Weizmann Institute, Rehovot.

Hoare, C.A.R. 1974. Monitors: an operating system structuring concept. In *CACM* 17(10), 549-557.

Hoare, C.A.R. 1978. Communicating Sequential Processes. In *CACM* 21(8), 666-677.

Hudak, P. 1986. Para-functional programming. In *IEEE Computing*, 19(8), 60-70.

Huntbach, M. 1987. The partial evaluation of Parlog programs. Research Report, Dept of Computing, Imperial College, London.

Ichiyoshi, N., Miyazaki, T. and Taki, K. 1987. A distributed implementation of Flat GHC on the Multi-PSI. In *Logic Programming: Proc. 4th Intl Conf.*, MIT Press, 257-275.

Inmos Limited, 1984. *Occam Programming Manual*. Prentice Hall, Englewood Cliffs, NJ.

Jones, S.B. 1984. A range of operating systems written in a purely functional style. Technical Monograph PRG-42, Oxford University Computing Laboratory.

Joseph, M., Prasad, V.R. and Narayana, K.T. 1984. *A Multiprocessor Operating System*. Prentice-Hall, Englewood Cliffs, NJ.

Kahn, G. and MacQueen, D.B. 1977. Coroutines and networks of parallel processes. In *Proc. IFIP 77*, Amsterdam: North-Holland, 993-998.

Kahn, K.M. and Miller, M.S. 1988. Language design and open systems. In *The Ecology of Computation*, North Holland.

Kawanobe, K. 1984. Current status and future plans of the Fifth Generation Computer Systems project. In *Proc. Intl Conf. on 5th Generation Computer Systems*, Amsterdam: North-Holland, 3-17.

Keller, R.M. and Lin, F. 1984. Simulated performance of a reduction-based multiprocessor. In *IEEE Computer*, 17(7), 70-82.

Komorowski, H.J. 1981. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. In *Proc. 9th ACM POPL Symp.*, Albuquequé, New Mexico.

Kowalski, R.A. 1974. Predicate logic as programming language. In *Information Processing 74; Proc. IFIP Congress*, Amsterdam: North-Holland, 569-574.

Kowalski, R.A. 1979. *Logic for Problem Solving*, New York: North-Holland.

Kowalski, R.A. 1983. Logic programming. In *Information Processing 83; Proc. IFIP 9th World Computer Congress 83*, Amsterdam: North-Holland, 133-145.

Kowalski, R.A. and Sergot, M.J. 1985. A logic-based calculus of events. In *New Generation Computing*, 4(1), 67-96.

Kusalik, A.J. 1986. Specification and initialization of a logic computer system. In New Generation Computing, 4(2), pp 189-209.

Lampson, B.W. and Sturgis, H. 1976. Crash recovery in a distributed data storage system. Technical report, Computer Science Laboratory, Xerox PARC, Palo Alto, CA.

McCabe, F.G., Foster I.T., Clark, K.L., Cheung, P. and Knowles, G. 1987. The X machine – a proposal for design and construction. Unpublished report, Dept of Computing, Imperial College.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I. 1965. *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, MA.

McDonald, C.S. 1987. fsh - a functional Unix command interpreter. In *Softw. P & E*, 17(1), 685-700.

Mierowsky,C., Taylor, S., Shapiro, E., Levy, J., and Safra, M. 1985. The design and implementation of Flat Concurrent Prolog. Technical Report CS85-09, Weizmann Institute, Rehovot, 1985.

Monteiro, L. 1984. A proposal for distributed programming in logic. In *Implementations of Prolog*, Ellis Horwood.

Moszkowski, B. 1984. Executing temporal logic programs. Technical report 55, Computer Laboratory, University of Cambridge, England.

Mueller, E.T, Moore, J.D. and Popek, G.J. 1983. A nested transaction mechanism for LOCUS. In *Proc. 8th Symp. Operating Systems Principles*, ACM SIGOPS.

Papadimitriou, C. 1986. *The Theory of Database Concurrency Control*. Computer Science Press.

Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. CACM 15(12), 1053-1058.

Periera, L.M. and Nasr, R. 1984. Delta-Prolog: a distributed logic programming language. In *Proc. Intl Conf. on 5th Generation Computer Systems*, Amsterdam: North-Holland, 283-291.

Peterson, J.L. and Silberschatz, A. 1983. *Operating System Concepts*. Addison-Wesley.

Pnueli, A. 1986. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, Springer-Verlag LNCS-224, 510-584.

Powell, M.L and Miller, B.P. 1983. Process migration in DEMOS/MP. In *Proc. 8th Symp. Operating Systems Principles*, ACM SIGOPS, 110-119.

Raphael, B. 1971. The frame problem in problem solving systems. In *Artificial Intelligence and Heuristic Programming*, Edinburgh University Press, Edinburgh, 159-169.

Reddy, U.S. 1985a. Narrowing as the operational semantics of functional programs. In *Proc. IEEE Symp. on Logic Programming*, 138-151, Boston, MA.

Reddy, U.S. 1985b. On the relationship between logic and functional languages. In *Logic Programming: Relations, Functions and Equations*, Prentice-Hall, Englewood Cliffs, NJ.

Richie, D.M. and Thompson, K. 1974. The Unix time-sharing system. In *CACM* 17(7), 365-375.

Ringwood, G.A. 1988. PARLOG86 and the dining logicians. In *CACM*, 31(1), 10-25.

Rokusawa, K., Ichiyoshi, N. and Chikayama, T. 1988. An efficient termination detection and abortion algorithm for distributed processing systems. Technical Report, ICOT, Tokyo.

Roussel, P. 1975. PROLOG: manuel de référence et d'utilisation. Groupe d'Intelligence Artificielle, Univerisite d'Aix-Marseille, Luminy.

Robinson, J.A. 1965. A machine oriented logic based on the resolution principle. In *JACM*, 12(1), 23-41.

Safra, M. and Shapiro, E. 1986. Metainterpreters for real. In *Proc. IFIP '86*.

Sato, H., Chikayama, T., Sugino, E. and Taki, K. 1987a. Outlines of PIMOS. In *Proc. 34th Annual Convention IPS Japan* (In Japanese).

Sato, M. and Sakurai, T. 1984. Qute: a functional language based on unification. In *Proc. Intl Conf. on 5th Generation Computer Systems*, Amsterdam: North-Holland, 157-165.

Sato, M., Shimizu, H., Matsumoto, A., Rokusawa, K. and Goto, A. 1987b. KL1 execution model for PIM cluster with shared memory. In *Logic Programming: Proc. 4th Intl Conf.*, MIT Press, 338-355.

Sergot M.J. 1982. A query-the-user facility for logic programming. In *New Horizons in Educational Computing*, Ellis Horwood.

Shapiro, E. 1983. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo.

Shapiro, E. 1984a. Systems programming in Concurrent Prolog. In *Proc. 11th ACM Symp. on Principles of Programming Languages*, New York: ACM, 93-105.

Shapiro, E. 1984b. Systolic programming: a paradigm for parallel processing. In *Proc. Intl Conf. on 5th Generation Computer Systems*, Amsterdam: North-Holland, 458-471.

Shapiro, E. 1986. Concurrent Prolog: a progress report. In *Fundamentals of Artificial Intelligence*, Springer-Verlag LNCS-232 , 277-313.

Shapiro, E. and Safra, M. 1986. Multiway merge with constant delay in Concurrent Prolog. In *New Generation Computing*, 4(2), 211-216.

Shultis, J. 1983. A Functional Shell. In *ACM SIGPLAN Notices*, 18(6), 202-11, 1983.

Silverman, W., Hirsch, M., Houri, A., and Shapiro, E. 1986. The Logix user manual, version 1.21. Technical Report CS-21, Weizmann Institute, Rehovot.

Sloman, M. and Kramer, J. 1986. *Distributed Systems and Computer Networks*, Prentice Hall.

Somogyi, Z. 1987. Stability of logic programs: how to connect don't-know nondeterministic logic programs to the outside world. Technical Report 87/11, Dept of Computer Science, University of Melbourne.

Stoye, W. 1984. A new scheme for writing functional operating systems. Research Report, University of Cambridge Computing Laboratory, Cambridge.

Swinehart, D.C., Zellweger, P.T., Beach, R.J. and Hagmann, R.B. 1986. A structural view of the Cedar programming environment. In *ACM Trans. Program. Lang. Syst.*, 8(4), 419-490.

Takagi, S., Yokoi, T., Uchida, S., Kurokawa, T., Hattori, T., Chikayama, T., Sakai, K. and Tsuji, J. 1984. Overall design of SIMPOS. In *Proc. 2nd Intl Logic Programming Conf.*, Uppsala, Sweden, 1-12.

Takeuchi, A. 1983. How to solve it in Concurrent Prolog. Unpublished note.

Takeuchi, A. and Furukawa, K. 1986. Parallel logic programming languages. In *Proc. 3rd Intl Logic Programming Conf.*, LNCS-225, Springer-Verlag, 242-254.

Tanenbaum, A.S. and van Renesse, R. 1985. Distributed operating systems. In *Computing Surveys*, 17(4), 419-470.

Taylor, S. 1987. Flat Concurrent Prolog initial performance information. Draft report, Weizmann Institute, Rehovot.

Taylor, S., Av-Ron, E. and Shapiro, E. 1987a. A layered method for process and code mapping. In *Journal of New Generation Computing*, 5(2), 185-205.

Taylor, S., Safra, S. and Shapiro, E. 1987b. A parallel implementation of Flat Concurrent Prolog. In *International Journal of Parallel Processing*, 15(3), 245-275.

Thom, J. and Zobel, J. (Eds.) 1986. Nu-Prolog reference manual. TR 86/10, Machine Intelligence Project, University of Melbourne.

Tick, E. and Warren, D.H.D. 1984. Towards a pipelined Prolog processor. In *New Generation Computing*, 2, 323-345.

Treleaven, P.C., Brownbridge, D.R. and Hopkins, R.P. 1982. Data-driven and demand-driven computer architecture. In *ACM Computing Surveys*, 14(1), 93-143.

Turner, D.A. 1981. The semantic elegance of applicative languages. In Proc. ACM *Conf. on Functional Programming Languages and Computer Architecture*, Portsmouth, NH, 85-92.

Uchida, S. 1987. Inference machines in FGCS project. In Proc. VLSI '87, IFIP TC-10, WG 10.5.

Ueda, K. 1986. *Guarded Horn Clauses*, EngD thesis, University of Tokyo. To be published by MIT press.

Ullman, J. 1982. *Principles of Database Systems*, Computer Science Press.

Vegdahl, S.R. 1984. A survey of proposed architectures for the execution of functional languages. In *IEEE Trans. Computers*, c-33(12), 1050-1071.

Wang, Y. and Morris, R. 1985. Load sharing in distributed computer systems. In *IEEE Trans. Computers*, C-34(3), 204-217.

Warren, D.H.D. 1987. Or-parallel execution models of Prolog. In *TAPSOFT '87, The 1987 Intl Jt Conf. on Theory and Practice of Software Development*, Pisa, Italy, Springer-Verlag, 243-259.

Warren, D.H.D., Periera, L.M. and Periera, F. 1977. Prolog - the language and its implementation compared with Lisp. In *ACM SIGPLAN Notices* 12(8).

Weihl, W. and Liskov, B. 1985. Implementation of resilient, atomic data types. In *ACM Trans. on Program. Lang. Syst.*, 7(2), 244-269.

Wirth, N. 1977. Design and implementation of Modula. In *Soft. P & E*, 7(4), 67-84.

Wulf, W.A., Levin, R. and Harbison, S.P. 1981. *HYDRA/C.mmp: An Experimental Computer System*, New York: McGraw-Hill.

Yang, R. and Aiso, H. 1986. P-Prolog: a parallel logic language based on exclusive relation. In *Proc. 3rd Intl Logic Programming Conf.*, NewYork: Springer-Verlag, 255-269.

15646